NARODOWA AGENCJA
WYMIANY AKADEMICKIEJ

Publikacja finansowana przez NAWA
w ramach działań komplementarnych do tych podejmowanych przez UJD w ramach sojuszu COLOURS

PODSTAWY BAZ DANYCH

PostgreSQL:
przykłady i ćwiczenia

Lidia Stępień

Marcin R. Stępień

Artur Gola

Uniwersytet Jana Długosza w Częstochowie

Podstawy baz danych
PostgreSQL: przykłady i ćwiczenia

Lidia Stępień, Marcin R. Stępień, Artur Gola

Częstochowa 2024

Recenzent
dr inż. Paweł Róg

Redaktor Naczelna Wydawnictwa
Paulina Piasecka-Florczyk

Redaktor statystyczny
Jarosław Kowalski

Korekta
Bożena Woźna-Szcześniak

Redakcja techniczna i projekt okładki
Bożena Woźna-Szcześniak

ISBN 978-83-67984-24-9

Wydawnictwo Naukowe Uniwersytetu Jana Długosza w Częstochowie
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl

Spis treści

Przedmowa

Baza danych dotyczy zawsze pewnego fragmentu rzeczywistości (obszaru analizy) i stanowi zbiór danych, posiadający określoną strukturę wewnętrzną, który reprezentuje ten fragment rzeczywistości i ułatwia przetwarzanie zgromadzonych danych. Do opisu struktury wewnętrznej bazy danych służą modele danych, pozwalające opisać własności danych w sposób ścisły (często sformalizowany) z wykorzystaniem języka matematyki, np. algebry relacji w modelu relacyjnym. Oprócz opisywania struktury danych model także określa dozwolone operacje na danych oraz sposób nakładania ograniczeń (więzów) zapewniających poprawność bazy danych.

Do obsługi bazy danych (rozumianej jako zbiór danych) wykorzystuje się system zarządzania bazą danych (SZBD, ang. *Database Management System*, DBMS). Jest on zorganizowanym zbiorem narzędzi umożliwiających realizację istotnych dla użytkownika operacji na danych. Dotychczas, w oparciu o różne modele, powstało wiele systemów zarządzania bazami danych, jednak wciąż najbardziej popularne pozostają te oparte o model relacyjny. Niezależnie od wyboru, oprogramowanie tego typu wymaga poznania podstaw teoretycznych modelu oraz języka wysokiego poziomu SQL (ang. *Structured Query Language*). Trudności nauki należy upatrywać w różnorodności i skomplikowaniu oprogramowania SZBD oferowanego przez różnych producentów. Autorzy tego podręcznika proponują naukę w oparciu o PostgreSQL jako jeden z najpopularniejszych otwartych systemów zarządzania relacyjnymi bazami danych (`https://db-engines.com/en/blog_post/106`). Oprogramowanie to od wielu lat nie spada w światowym rankingu popularności SZBD. W roku 2023 PostgreSQL zajął pierwsze miejsce w rankingu, a od wielu lat ciągle pozostaje w pierwszej dziesiątce wśród prawie 400 wykorzystywanych SZBD wszystkich typów (`https://db-engines.com/en/`).

Proponowany podręcznik zawiera wprowadzenie do podstawowych instrukcji języka SQL wykonywanych w relacyjnych bazach danych bogato ilustrowanych przykładami oraz zadania do samodzielnego rozwiązania utrwalające nabyte umiejętności. Czytelnik zostanie zapoznany ze sposobem tworzenia tabel (podstawowych struktur danych w modelu relacyjnym), ich modyfikowania, usuwania oraz manipulowania danymi. Pokażemy, jak stosując indeksy, polepszyć czas wykonania zapytania, omawiając przy tym wybrane pod-

stawowe zagadnienia związane z funkcjami języka proceduralnego `plpgSQL`. W ostatnim rozdziale przedstawimy sposób tworzenia aplikacji bazodanowej w języku Java, przy czym zakładamy, że czytelnik opanował już umiejętności programowania w tym języku.

Rozdział 1

Wprowadzenie do systemu PostgreSQL

W tym rozdziale zostanie opisana architektura systemu zarządzania relacyjnymi bazami danych PostgreSQL, instalacja systemu w systemach operacyjnych Windows oraz Linux oraz polecenia klienta `psql`. Przedstawione zostaną również podstawowe typy danych i operatory używane w języku SQL.

1.1. Architektura systemu

PostgreSQL jest jednym z bardziej popularnych systemów zarządzania bazami danych i jednym z niewielu systemów zarządzania oferującym obiektowo-relacyjne podejście do baz danych. Architektura systemu to architektura `klient-serwer`. W tym modelu jeden klient może jednocześnie korzystać z usług wielu serwerów, a jeden serwer być użytkowany przez wielu użytkowników w tym samym czasie. Oznacza to podział ról pomiędzy klienta a serwer w następujący sposób:

- Klient jest stroną żądającą dostępu do bazy danych – wysyła żądanie do serwera i oczekuje na jego odpowiedź, wykorzystując w tym celu dedykowany interfejs. Nie ma jednak wpływu na połączenie z serwerem, ochronę znajdujących się na nich danych, ani na ich przetwarzanie.
- Po stronie serwera jest ustanowienie połączenia, przetwarzanie danych oraz ich ochrona. Serwer pełni rolę pasywną – świadczy usługę (dostarcza przetworzone w odpowiedni sposób dane) na żądanie.

Dla PostgreSQL stosowane są dwa narzędzia typu klient - graficzny interfejs użytkownika `pgAdmin` i uruchamiane w wierszu poleceń narzędzie `psql`. W tym podręczniku korzystać będziemy z `psql` – terminalowego klienta pozwalającego na połączenie z bazą danych

PostgreSQL oraz na wykonywanie zapytań SQL w sposób interaktywny lub z wcześniej przygotowanego pliku z poleceniami do wykonania. Program ten można również wykorzystać do zapisywania wyników zapytań do plików.

1.2. Przygotowanie środowiska

Przed rozpoczęciem dokładnej lektury tego podręcznika należy przygotować określone oprogramowanie i narzędzia. W kolejnych podrozdziałach przedstawimy, jak to zrobić w najpopularniejszych systemach operacyjnych Windows i Linux.

1.2.1. Instalowanie systemu PostgreSQL dla systemu Windows

W celu pobrania systemu PostgreSQL dla Windowsa należy:

1. Wejść na stronę `https://www.postgresql.org/download/` i wybrać opcję *Windows* z listy *Packages and Installers*.
2. Następnie po wybraniu opcji *Download the Installer*, pobieramy odpowiednią wersję systemu PostgreSQL (w tym podręczniku używana będzie wersja 15.8).
3. Uruchamiamy interaktywny instalator i w większości kroków wciskamy przycisk *Next*. Gdy pojawi się prośba o wskazanie katalogu dla danych, warto jest podać ścieżkę, którą można łatwo zapamiętać.
4. Podajemy hasło dla administratora (domyślnie jest to użytkownik *postgres*).
5. Nie zmieniamy domyślnie używanego portu, chyba że powoduje on konflikt z aplikacją, która już jest zainstalowana w systemie.
6. W pozostałych krokach wciskamy przycisk *Next* i czekamy na zakończenie instalacji.

W celu sprawdzenia, czy zmienna *Path* została poprawnie ustawiona, należy uruchomić wiersz poleceń, wpisać instrukcję: `psql -U postgres` i wcisnąć *Enter*.

Jeśli pojawi się błąd: *'psql' is not recognized as an internal or external command, operable program or batch file*, należy do zmiennej *Path* dodać katalog z binariami systemu PostgreSQL.

W tym celu należy wykonać następujące kroki:

1. W polu wyszukiwania w systemie Windows wpisujemy nazwę *zmienne środowiskowe*.
2. Następnie wybieramy opcję *Zmienne środowiskowe*, zaznaczamy zmienną *Path* i wciskamy przycisk *Edycja*.
3. Wybieramy przycisk *Nowy*.
4. W eksploratorze plików odnajdujemy katalog, w którym zainstalowany jest system PostgreSQL i dodajemy ścieżkę do katalogu *bin* z instalacji systemu PostgreSQL.
5. Zatwierdzamy przyciskiem *OK* i restartujemy system.

Po ponownym uruchomieniu komputera, w wierszu poleceń wpisujemy instrukcję:

```
psql -U postgres
```

i zatwierdzamy ją klawiszem *Enter*.

Uruchomienie powłoki systemu PostgreSQL wymaga podania hasła ustawionego w kroku 4 instalacji systemu, wciśnięcia klawisza *Enter*, co sygnalizowane jest zmianą znaku zachęty. Wyjście z powłoki następuje poprzez wpisanie polecenia `\q` i zatwierdzenie go klawiszem *Enter*.

1.2.2. Instalowanie systemu PostgreSQL dla systemu Linux

Instalacja systemu PostgreSQL w systemie Ubuntu lub w systemach bazujących na dystrybucji Debian:

1. Należy wykonać instrukcje zgodnie z poleceniami instalacyjnymi dostępnymi na stronie `https://www.postgresql.org/download/linux/ubuntu/`
2. W trakcie instalacji systemu PostgreSQL zostanie utworzony użytkownik `postgres`, dzięki czemu możliwe będzie uruchomienie powłoki systemu dla tego użytkownika:

   ```
   sudo su postgres
   ```

   Zmiana zachęty zasygnalizuje dostęp do powłoki, gdzie będzie można uruchomić klienta zatwierdzając instrukcję `psql`.

Instrukcje dotyczące instalacji systemu PostgreSQL w innych dystrybucjach znajdują się w dokumentacji systemu. Strona pobierania systemu PostgreSQL dla systemów linuksowych znajduje się pod adresem `https://www.postgresql.org/download/`.

1.3. Polecenia klienta `psql`

Tabela 1.1: Wybrane polecenia klienta `psql`

Polecenie	Opis
`\c[onnect] [DBNAME]`	połączenie się z bazą danych, np. `\c ksiegarnia;`
`\h [NAME]`	pomoc dotycząca składni poleceń SQL, * dla wszystkich poleceń
`\q`	wyjście z psql
`\e [FILE]`	edycja bufora zapytań (lub pliku) za pomocą zewnętrznego edytora
`\ef [FUNCNAME]`	edycja definicji funkcji za pomocą zewnętrznego edytora
`\p`	pokazanie zawartości bufora zapytań
Ciąg dalszy tabeli na następnej stronie	

Ciąg dalszy tabeli	
Polecenie	**Opis**
`\r`	reset (czyszczenie) bufora zapytań
`\s [FILE]`	wyświetlenie historii lub zapisanie jej do pliku
`\w FILE`	zapis bufora zapytań do pliku
`\echo [STRING]`	wypisanie ciągu znaków na standardowe wyjście
`\i FILE`	wykonanie poleceń z pliku
(opcje: S – pokaż obiekty systemowe, + – dodatkowy szczegół)	
`\d[S+]`	wyświetla tabele, widoki i sekwencje
`\d[S+] NAME`	opisuje tabelę, widok, sekwencję lub indeks
`\da[+] [PATTERN]`	wyświetla agregacje
`\db[+] [PATTERN]`	wyświetla przestrzenie tabel
`\dc[S] [PATTERN]`	wyświetla konwersje
`\dC [PATTERN]`	wyświetla konswersje typów
`\dd[S] [PATTERN]`	wyświetla komentarze w obiekcie
`\dD[S] [PATTERN]`	wyświetla dziedziny
`\des[+] [PATTERN]`	wyświetla serwery zewnętrzne
`\deu[+] [PATTERN]`	wyświetla mapowania użytkowników
`\dew[+] [PATTERN]`	wyświetla wrappery danych zwenętrznych
`\df[antw][S+] [PATRN]`	wyświetla funkcje (tylko agregujące/standardowe-/triggery/okna)
`\dF[+] [PATTERN]`	wyświetla konfiguracje wyszukiwania tekstu
`\dFd[+] [PATTERN]`	wyświetla słowniki wyszukiwania tekstu
`\dFp[+] [PATTERN]`	wyświetla parsery wyszukiwania tekstu
`\dFt[+] [PATTERN]`	wyświetla szablony wyszukiwania tekstu
`\dg[+] [PATTERN]`	wyświetla role (grupy)
`\di[S+] [PATTERN]`	wyświetla indeksy
`\dl`	wyświetla duże obiekty, to samo co `\lo_list`
`\dn[+] [PATTERN]`	wyświetla schematy
`\do[S] [PATTERN]`	wyświetla operatory
`\dp [PATTERN]`	wyświetla uprawnienia dostępu do tabel/perspektyw i sekwencji
`\ds[S+] [PATTERN]`	wyświetla sekwencje
`\dt[S+] [PATTERN]`	wyświetla tabele
`\dT[S+] [PATTERN]`	wyświetla typy danych
`\du[+] [PATTERN]`	wyświetla role (użytkowników)
`\dv[S+] [PATTERN]`	wyświetla perspektywy
Ciąg dalszy tabeli na następnej stronie	

Ciąg dalszy tabeli	
Polecenie	**Opis**
`\l[+]`	wyświetla wszystkie bazy danych
`\z [PATTERN]`	to samo co `\dp`
`\encoding [ENCODING]`	pokazuje lub ustawia kodowanie klienta
`\password [USERNAME]`	bezpieczna zmiana hasła dla użytkownika
`\prompt [TEXT] NAME`	wyświetla komunikat przy zmianie zmiennej wewnętrznej
`\set [NAME [VALUE]]`	ustala zmienną wewnętrzną lub wyświetla wszystkie, jeśli brak parametrów
`\unset NAME`	usuwa zmienną wewnętrzną

1.4. Funkcje systemu informatycznego

Tabela 1.2: Wybrane funkcje systemu informatycznego

Nazwa	**Typ zwracany**	**Opis**
current_catalog	name	nazwa bieżącej bazy danych (zwanej „katalogiem” w standardzie SQL)
current_database()	name	nazwa bieżącej bazy danych
current_schema[()]	name	nazwa bieżącego schematu
current_schemas(boolean)	name	nazwy schematów w ścieżce poszukiwać, opcjonalnie zawierającej ukryte schematy
current_user	name	nazwa użytkownika bieżącego kontekstu wykonania (bieżącej sesji)
current_query	text	tekst aktualnie wykonywanego zapytania, przesłanego przez klienta (może zawierać więcej niż jedno zapytanie)
pg_backend_pid()	int	Identyfikator procesu serwera przypisanego do bieżącej sesji
inet_client_addr()	inet	Adres połączenia zdalnego

Ciąg dalszy tabeli na następnej stronie

Ciąg dalszy tabeli		
Nazwa	**Typ zwracany**	**Opis**
inet_client_port()	int	port połączenia zdalnego
inet_server_addr()	inet	adres lokalnego połączenia
inet_server_port()	int	port lokalnego połączenia
pg_my_temp_schema()	oid	OID tymczasowego schematu sesji lub 0, jeśli brak
pg_is_other_temp_schema(oid)	boolean	Czy schemat jest tymczasowym schematem innej sesji?
pg_postmaster_start_time()	timestamp with time zone	czas uruchomienia serwera
pg_conf_load_time()	timestamp with time zone	czas wczytania konfiguracji
session_user	name	nazwa użytkownika sesji równoważna z current_user
version()	text	informacje o wersji PostgreSQL

1.5. Komentarze

W PostgreSQL dostępne są dwa rodzaje komentarzy:

`--` jednowierszowy oraz

`/* */` wielowierszowy.

1.6. Typy danych i operatory języka SQL

Relacyjne bazy danych oraz ich podstawowy język SQL to dwie główne technologie służące do przechowywania i przetwarzania dużych zbiorów danych. Pomimo obowiązującego standardu ANSI producenci proponują własne typy danych, często na przykład wprowadzając własne synonimy. W tym podręczniku opierać się będziemy na typach danych i operatorach języka SQL obowiązujących w systemie PostgreSQL (dokumentacja pod linkiem: `https://www.postgresql.org/docs/`).

1.6.1. Typy danych

Tabela 1.3: Typy danych PostgreSQL

Nazwa	Alias	Opis
bigint	int8	8-bajtowa liczba całkowita od −9223372036854775808 do 9223372036854775807
bigserial	serial8	8-bajtowa liczba naturalna z automatycznym przyrostem od 1 do 9223372036854775807
bit [(n)]		łańcuch bitów o stałej długości, np. BIT(3) → B'101'
bit varying [(n)]	varbit	łańcuch bitów o zmiennej długości
boolean	bool	wartość logiczna (true/false)
box		prostokąt na płaszczyźnie, 32 bajty, ((x1,y1),(x2,y2))
byte		dane binarne ("byte array")
character varying [(n)]	varchar [(n)]	łańcuch znaków o zmiennej długości
character [(n)]	char [(n)]	łańcuch znaków o stałej długości
cidr		IPv4 lub IPv6 adres sieciowy, 7 lub 19B
circle		okrąg na płaszczyźnie, 24 bajty, `<(x,y),r>` środek i promień
date		data kalendarzowa (rok, miesiąc, dzień), 4 bajty, od 4713BC do 5874897AD, dokł. 1 dzień
double precision	float8	liczba zmiennoprzecinkowa o podwójnej precyzji (8 bajtów) 15 cyfr dziesiętnych
inet		IPv4 lub IPv6 adres hosta, 7 lub 19B
integer	int, int4	4-bajtowa liczba całkowita
interval [fields] [(p)]		przedział czasu, 12 bajtów, dokł. 1μ, od −178000000 lat do 178000000 lat
line		linia nieskończona na płaszczyźnie, 32 bajty, ((x1,y1),(x2,y2))
lseg		odcinek linii na płaszczyźnie, 32 bajty, ((x1,y1),(x2,y2))

Ciąg dalszy tabeli na następnej stronie

Ciąg dalszy tabeli		
Nazwa	**Alias**	**Opis**
macaddr		MAC (ang. *Media Access Control*) adres, 6B
money		waluta
numeric [(p,s)]	decimal[(p,s)]	dokładna liczba o wybranej precyzji
path		ścieżka geometryczna na płaszczyźnie ((x1,y1),...)
point		punkt geometryczny na płaszczyźnie 16 bajtów, (x,y)
polygon		zamknięta ścieżka geometryczna na płaszczyźnie ((x1,y1),...)
real	float4	liczba zmiennoprzecinkowa o pojedynczej precyzji (4 bajty) precyzja 6 cyfr dziesiętnych
smallint	int2	2-bajtowa liczba całkowita od −32768 do 32767
serial	serial4	4-bajtowa liczba naturalna z automatycznym przyrostem od 1 do 2147483647
text		łańcuch znaków o zmiennej długości
time [(p)] [without time zone]		czas dnia (bez strefy czasu), 8 bajtów, od 00:00:00 do 24:00:00, dokładność 1μs, 14 cyfr
time [(p)] [with time zone]	timetz	czas dnia (ze strefą czasu), 12 bajtów, od 00:00:00+1459 do 24:00:00-1459 dokładność 1μs, 14 cyfr
timestamp [(p)] [without time zone]		data i czas (bez strefy czasu), 8 bajtów, od 4713BC do 294276AD, dokładność 1μs, 14 cyfr
timestamp [(p)] with time zone	timestamptz	data i czas (ze strefą czasu), 8 bajtów, od 4713BC do 294276AD, dokładność 1μs, 14 cyfr
tsquery		tekst szukania zapytania
tsvector		tekst szukania dokumentu
txid_snapshot		transakcja na poziomie użytkownika ID

Ciąg dalszy tabeli na następnej stronie

Ciąg dalszy tabeli		
Nazwa	**Alias**	**Opis**
uuid		uniwersalny, unikalny identyfikator (ang. *universally unique identifier*)
xml		dane w formacie XML (ang. *Extensible Markup Language data*)

1.6.2. Operatory matematyczne

Tabela 1.4: Operatory matematyczne PostgreSQL

Operator	Opis	Przykład	Wynik
+	dodawanie	2 + 2	4
-	odejmowanie	2 - 3	-1
*	mnożenie	2 * 3	6
/	dzielenie (całkowite dzielenie obcina wynik)	4 / 2	2
%	modulo (reszta)	5%4	1
^	podnoszenie do potęgi	2.0^3.0	8
|/	pierwiastek kwadratowy	|/25.0	5
||/	pierwiastek sześcienny	||/27.0	3
!	silnia	5!	120
!!	silnia (prefix operator)	!!5	120
@	wartość bezwzględna	@ – 5.0	5
&	bitowy AND	91&15	11
|	bitowy OR	32|3	35
#	bitowy XOR	17#5	20
~	bitowy NOT	~ 1	-2
<<	bitowe przesunięcie w lewo	1 << 4	16
>>	bitowe przesunięcie w prawo	8 >> 2	2

1.6.3. Operatory logiczne i porównania

Operatory porównania wykorzystywane w PostgreSQL: < (mniejszy), > (większy), <= (mniejszy lub równy), >= (większy lub równy), = (równy), <> lub ! = (różny).

Tabela 1.5: Tabela prawdy dla negacji (**NOT**)

a	**NOT a**
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Tabela 1.6: Tabele prawdy dla koniunkcji (**AND**) i alternatywy (**OR**)

a	**b**	**a AND b**	**a OR b**
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

Specjalne operatory służące do sprawdzenia czy wyrażenie jest jedną z wyróżnionych wartości:

- czy wyrażenie jest/nie jest wartością nieokreśloną `NULL`:

```
<wyrażenie> IS NULL
<wyrażenie> IS NOT NULL
```

- czy wyrażenie jest różne/lub nie od wskazanego we frazie `FROM` wyrażenia:

```
<wyrażenie> IS DISTINCT FROM <wyrażenie>
<wyrażenie> IS NOT DISTINCT FROM <wyrażenie>
```

- czy wyrażenie ma wartość `TRUE` lub `FALSE`:

```
<wyrażenie> IS TRUE
<wyrażenie> IS NOT TRUE
<wyrażenie> IS FALSE
<wyrażenie> IS NOT FALSE
```

- czy wyrażenie jest nieznane lub nie:

```
<wyrażenie> IS UNKNOWN
<wyrażenie> IS NOT UNKNOWN
```

1.6.4. Symbole i operatory służące do budowania wyrażeń regularnych

`Wyrażenia regularne` (ang. *regular expressions, regex*) są używane do wyszukiwania określonych ciągów znaków w tekstach, które spełniają zdefiniowane w wyrażeniu regularnym zasady. Tworzą wzorce, do budowy których wykorzystuje się operatory oraz symbole przedstawione odpowiednio w tabelach 1.7 oraz 1.8.

Tabela 1.7: Operatory wyrażeń regularnych

Operator	Użycie	Opis
$\sim$	`'ciąg'` $\sim$ `'regex'`	Porównanie wyrażenia regularnego, dające wynik *prawda*, jeżeli wyrażenie **zgadza się** z ciągiem
$!\sim$	`'ciąg'` $!\sim$ `'regex'`	Porównanie wyrażenia regularnego, dające wynik *prawda*, jeżeli wyrażenie **nie zgadza się** z ciągiem
$\sim*$	`'ciąg'` $\sim*$ `'regex'`	Porównanie wyrażenia regularnego bez rozróżniania wielkości liter, dające wynik *prawda*, jeżeli wyrażenie **zgadza się** z ciągiem
$!\sim*$	`'ciąg'` $!\sim*$ `'regex'`	Porównanie wyrażenia regularnego bez rozróżniania wielkości liter, dające wynik *prawda*, jeżeli wyrażenie **nie zgadza się** z ciągiem

Tabela 1.8: Symbole w wyrażeniach regularnych

Symbol	Zastosowanie	Opis
^	^`wyrażenie`	Odpowiada początkowi ciągu znakowego
$	`wyrażenie`$	Odpowiada końcowi ciągu znakowego
.	.	Odpowiada dowolnemu pojedynczemu znakowi
[ ]	[*abc*]	Odpowiada każdemu pojedynczemu znakowi, który wymieniono w nawiasach (tu `a`, `b` lub `c`)
[^]	[^*abc*]	Odpowiada każdemu pojedynczemu znakowi, innemu niż znajdujące się w nawiasach (tu inne niż `a`, `b` lub `c`)
Ciąg dalszy tabeli na następnej stronie		

Ciąg dalszy tabeli		
Symbol	**Zastosowanie**	**Opis**
`[-]`	$[a-z]$	Odpowiada każdemu znakowi z zakresu znaków umieszczonych w nawiasach i oddzielonych myślnikiem (tu znaki od `a` do `z`)
[^-]	[^a-z]	Odpowiada każdemu znakowi spoza zakresu znaków umieszczonych w nawiasach i oddzielonych myślnikiem (tu z innego zakresu niż od `a` do `z`)
?	$a?$	Oznacza zero lub jedno wystąpienie znaku bądź poprzedzającej pytajnik sekwencji wyrażenia regularnego
*	$a*$	Oznacza zero lub kilka wystąpień znaku bądź poprzedzającej gwiazdkę sekwencji wyrażenia regularnego
+	$a+$	Oznacza jedno lub kilka wystąpień znaku bądź poprzedzającej plus sekwencji wyrażenia regularnego
|	$wyr1\|wyr2$	Oznacza wyrażenie z lewej lub z prawej strony znaku | (tu `wyr1` albo `wyr2`)
()	$(wyr1)wyr2$	Jawnie grupuje wyrażenia w celu określenia priorytetu specjalnych symboli znakowych
{}	$\{m\}$ $\{m,\}$ $\{m,n\}$	sekwencja dokładnie `m` znaków sekwencja co najmniej `m` znaków zakres liczby znaków, przy czym $m < n$
$\backslash d$	$\backslash d\{2\}$	dowolna cyfra (tu dokładnie dwie cyfry)

Rozdział 2

Podstawowe operacje na tabelach

Relacyjny model danych został stworzony w 1970 roku przez Edgara F. Codda w oparciu o algebrę relacji. W modelu tym dane są uporządkowane za pomocą relacji jako zbiory krotek, w których krotka jest zestawem atrybutów uporządkowanych w określonej kolejności. Częściej stosuje się dla krotki zamienną nazwę „rekord". Dla relacji będącej podstawową strukturą danych w relacyjnym modelu danych często stosuje się zamienną, potoczną nazwę „tabela".

W tym podręczniku będziemy rozważać przykładową relacyjną bazę danych reprezentującą sprzedaż książek w księgarni internetowej o nazwie „ksiegarnia". W bazie tej w relacji `t_klient` przechowywane są między innymi informacje o klientach. Atrybuty, opisujące każdy rekord, obejmują informacje o pojedynczym kliencie, takie między innymi jak imię i nazwisko klienta, adres dostawy oraz telefon kontaktowy. Przykładowe dane zawarte w relacji `t_klient` przedstawione zostały w tabeli 2.1.

Tabela 2.1: Przykładowe dane przechowywane w relacji *t_klient*

id klienta	imie	nazwisko	...	telefon
1	Jan	Kowalski	...	624-455-566
2	Tadeusz	Malinowski	...	624-424-332
3	Krystyna	Torbicka	...	624-212-111

Każda relacja jest dwuwymiarową tabelą, w której wiersze są rekordami, a atrybuty je opisujące są kolumnami. W obrębie tabeli rekordy nie mogą się powtarzać. Stąd, choć technicznie nie jest to wymagane, większość tabel w relacyjnym modelu danych posiada kolumnę (lub grupę kolumn) nazywaną *kluczem głównym* (często również określanym jako *klucz podstawowy*) (ang. *primary key*), która w unikatowy sposób identyfikuje pojedynczy wiersz. W tabeli 2.1 kolumna `id_klienta` jest jej wyróżnionym kluczem głównym.

Dane w obrębie kolumny należą do tej samej dziedziny i posiadają typ danych opisujący rodzaj przechowywanych w niej danych. Na rysunku 2.1 przedstawiono graficznie w postaci diagramu ERD (ang. *Entity-Relationship Diagram*) tabele rozważanej bazy danych `ksiegarnia` wraz z ich zależnościami (referencyjnymi ograniczeniami integralnościowymi - potocznie zwanymi „relacjami").

Rysunek 2.1: Diagram EDR bazy danych `ksiegarnia`

Większość operacji w relacyjnych bazach danych i wszystkich systemach zarządzania danymi jest związana z tabelami i danymi w nich przechowywanymi. Ogólnie operacje te można podzielić na cztery grupy: tworzenie, odczyt, aktualizowanie i usuwanie danych. Aby operować na danych, trzeba najpierw przygotować definicję zbioru danych, uzupełnić zbiór rekordami z danymi, aby następnie móc je modyfikować, odczytywać, aktualizować,

a gdy są już niepotrzebne również usuwać. Te operacje są zwykle nazywane operacjami CRUD (ang. *Create, Read, Update, Delete*), co wynika z kolejności wykonywania każdej operacji w opisanym wyżej cyklu życia zbioru danych.

W relacyjnych bazach danych operacje CRUD są wykonywane za pomocą SQL-a pod kontrolą systemu zarządzania bazami danych. Poszczególne SZBD nieco różnią się sposobem operowania danymi, a nawet składnią SQL-a. Istnieje standard ANSI (ang. *American National Standards Institute*) opisujący SQL, który w dużym stopniu przestrzegany jest przez SZBD. Jednak każdy z dostępnych obecnie na rynku SZBD posiada pewne specyficzne interpretacje i rozszerzenia tego standardu, które odróżniają je od siebie.

W tym i dalszych rozdziałach czytelnik zapoznany zostanie ze wszystkimi instrukcjami SQL-a, często w skróconej postaci, odpowiadającymi wymienionym wyżej operacjom. Natomiast po pełen opis instrukcji języka SQL zainteresowanego czytelnika odsyłamy do dokumentacji PostgreSQL znajdującej się pod linkiem: `https://www.postgresql.org/docs/15/index.html`.

Pisząc instrukcje w języku SQL, można, ale nie jest to wymóg, dostosować się do podanych niżej, ogólnie przyjętych zasad ułatwiających zarówno pisanie poleceń, ich kompilację, jak i poprawiających czytelność kodu.

- Polecenia SQL mogą być rozmieszczone w kilku wierszach.
- Dobrym zwyczajem jest umieszczanie klauzul od nowych linii.
- Koniec polecenia SQL musi być zakończy średnikiem.
- Nie wolno dzielić słowa pomiędzy wiersze.
- Dozwolone jest używanie małych lub wielkich liter, chyba że zależy nam na rozróżnianiu ich wielkości.
- Baza PostgreSQL jest czuła na wielkość liter używanych w identyfikatorach (nazwy tabel, kolumn itp.). W przypadku gdy nazwa została zapisana z dużymi literami w zapytaniu SQL musi być objęta znakami cudzysłowów.

W prezentowanej w tym podręczniku składni instrukcji języka SQL używane będą nawiasy kwadratowe ([]) dla oznaczenia elementów opcjonalnych oraz nawiasy klamrowe ({ }) dla oznaczenia zbioru możliwych do wyboru opcji rozdzielonych kreską pionową (|).

2.1. Tworzenie tabeli

Do tworzenia nowej, pustej tabeli w bazie danych służy polecenie `CREATE TABLE`.

```
CREATE TABLE [IF NOT EXISTS] table_name (
  column_name data_type [DEFAULT default_expr][column_constraint[...]]
  [, ... ]
  | table_constraint
) [TABLESPACE tablespace];
```

gdzie:

- `column_constraint`:

```
[CONSTRAINT constraint_name]
{NOT NULL | NULL | UNIQUE index_parameters |
PRIMARY KEY index_parameters |
CHECK (expression) | REFERENCES reftable [(refcolumn)]
[ON DELETE action][ON UPDATE action]}
```

- `table_constraint`:

```
[CONSTRAINT constraint_name]
{UNIQUE (column_name [,...]) index_parameters |
PRIMARY KEY (column_name [,...]) index_parameters |
CHECK (expression) |
FOREIGN KEY (column_name [,...] )
REFERENCES reftable [(refcolumn [,...])]
[ON DELETE action][ON UPDATE action]}
```

- `index_parameters` w ograniczeniach `UNIQUE` i `PRIMARY KEY`:

```
[WITH (storage_parameter [=value] [,...])]
[USING INDEX TABLESPACE tablespace]
```

Użycie opcji `IF NOT EXISTS` w poleceniu `CREATE TABLE` powoduje pominięcie wykonywania instrukcji w sytuacji, gdy tabela o podanej nazwie już istnieje w bazie danych. Brak tej opcji spowoduje wystąpienie błędu: `ERROR: relation...already exists`. Nazwa tabeli `table_name` musi być unikatowa w bazie danych. Warunek unikatowości dotyczy również nazw kolumn `column_name` w tworzonej tabeli. Każda tworzona kolumna musi mieć przyporządkowany jeden z podstawowych typów danych z systemu PostgreSQL (patrz tabela 1.3 na stronie 14) takich, jak typy:

- liczbowe;
- znakowe – zwracamy uwagę na różnicę pomiędzy typem znakowym stałej długości `char(n)` dopełnianym znakami spacji do maksymalnej długości n znaków oraz zmiennej długości `varchar(n)` dopasowującym się do rzeczywistej długości łańcucha, przy czym nie przekraczającym maksymalnej, zadeklarowanej długości n;
- logiczne;
- daty i godziny – traktowane jako łańcuch;
- struktury danych (w formacie JSON i tablice).

Definiowane w nowo tworzonej tabeli kolumny mogą mieć zdefiniowane ograniczenia integralnościowe `column_constraint`, które określają specjalne cechy kolumn i gwarantują, że wszystkie wiersze z danej kolumny będą zgodne ze zdefiniowanym warunkiem, między innymi takim jak:

- `imie varchar(20) NOT NULL` – atrybut obowiązkowy;
- `cena numeric(4,2) CHECK(cena > 0)` – nie pozwala na wstawienie do kolumny `cena` kwoty, która nie jest dodatnia;
- `st varchar(7) CHECK(st in ('inż.','mgr','dr','dr hab.'))` – dopuszczalne są dla stopni naukowych tylko te wymienione w zbiorze;
- `data_zam date check(data_zam <= current_date)` – data zamówienia nie może być późniejsza niż bieżąca data.

W budowaniu warunków ograniczeń wykorzystuje się między innymi operatory: =, <, ! =, `is null`, `like` ('%' – łańcuch o długości >= 0; '_' - pojedynczy znak), `between...and`, `in (...)`, `and`, `or`, `not`. Pełna lista dostępnych w PostgreSQL operatorów znajduje się w podrozdziałach 1.4 oraz 1.6.3. Ponadto ograniczeniom można nadawać unikatową nazwę `constraint_name`, dzięki czemu łatwiejsze jest manipulowanie tymi ograniczeniami. Samodzielnie nienazwanym ograniczeniom PostgreSQL nada własne nazwy, które między innymi można odczytać, wydając w kliencie `psql` polecenie `\d nazwa_tabeli`. Pełną listę poleceń klienta `psql` można zobaczyć w podrozdziale 1.1 na stronie 10.

Przedstawimy teraz polecenia tworzące dwie przykładowe tabele z bazy danych o nazwie `ksiegarnia` pokazanej na diagramie ERD na rysunku 2.1. Ze względu na definiowane w obrębie tabel klucze obce (ang. *foreign key*), kolejność tworzenia tabel jest istotna. Jako pierwsze powinny zostać zdefiniowane wszystkie te tabele, które posiadają tylko klucze główne. Następnie te, w których znajduje się poza kluczem głównym klucz obcy, gdyż kompilator sprawdza, czy istnieje tabela i kolumna, do której tworzona jest referencja. Na końcu tabele składające się tylko z kluczy obcych (tzw. *tabele asocjacyjne* - łącznikowe).

Pierwszą tworzoną tabelą będzie tabela o nazwie `t_klient`, w której znajdują się między innymi całkowitoliczbowa kolumna `id_klienta`, która jest jej kluczem głównym oraz kolumny znakowe `kod` oraz `telefon`, dla których zostały zdefiniowane ograniczenia dla wprowadzanych danych z użyciem wyrażeń regularnych.

```
CREATE TABLE t_klient (
  id_klienta int2 PRIMARY KEY,
  imie varchar(30),
  nazwisko varchar(40),
  ulica varchar(30),
  miasto varchar(30),
  wojewodztwo varchar(30),
  kod char(6) CHECK(kod ~ ('^\d{2}-\d{3}$')),
  telefon char(11) CHECK(telefon ~ ('^\d{3}-\d{3}-\d{3}$'))
);
```

W kolumnie `kod` dopuszczalne są tylko ciągi znaków zaczynające się od dwóch znaków cyfr (^\d{2}), po których następuje znak minusa (-) i kończące się trzema znakami cyfr (\d{3}$). W warunku użyto operatora porównania dla wyrażeń regularnych, czyli tyldy (∼). W przypadku numeru telefonu format wprowadzanych danych wymaga, aby cyfry pogrupowane zostały po 3 i rozdzielone znakiem minusa (-) (na początku ^\d{3}, w środku \d{3}, na końcu \d{3}$).

Drugi przykład przedstawia polecenie tworzenia tabeli `t_zamowienie`, w której zdefiniowany został klucz główny `id_zam`. Tworzone jest również ograniczenie integralnościowe o nazwie `klucz_od_klienta`, które definiuje klucz obcy na kolumnie `id_klienta` tabeli jako referencję do tabeli `t_klient` i zdefiniowanego w niej klucza głównego `id_klienta` z kaskadowym usuwaniem. Przy usuwaniu kaskadowym rekordu z tabeli z kluczem podstawowym usuwane są automatycznie rekordy z relacji, w której wartość klucza obcego jest równa wartości klucza podstawowego usuwanego rekordu. Należy zatem ostrożnie korzystać z tej opcji, aby nie utracić danych powiązanych z usuwanym kluczem głównym. Jeśli nie zostanie użyte kaskadowe usuwanie tabel, operacja usuwania tabel musi przebiegać w odwrotnej kolejności do ich tworzenia.

```
CREATE TABLE t_zamowienie (
  id_zam int2 PRIMARY KEY,
  id_klienta int2,
  data_zam DATE DEFAULT current_date,
  karta int2,
  zrealizowane int2,
  CONSTRAINT klucz_od_klienta FOREIGN KEY(id_klienta)
  REFERENCES t_klient(id_klienta) ON DELETE CASCADE
);
```

2.2. Modyfikowanie struktury tabeli

Po utworzeniu tabeli można dowolnie modyfikować jej strukturę, używając w tym celu polecenia `ALTER TABLE`:

```
ALTER TABLE [ONLY] name [*]
    action [,...];
```

gdzie `action` to:

- dodanie nowej kolumny:

  ```
  ADD [COLUMN] column type [column_constraint [...]]
  ```

- usunięcie kolumny:

  ```
  DROP [COLUMN] column [RESTRICT|CASCADE]
  ```

- modyfikacja typu kolumny:

  ```
  ALTER [COLUMN] column [SET DATA] TYPE type [USING expression]
  ```

- dodanie wartości domyślnej dla danych w kolumnie:

  ```
  ALTER [COLUMN] column SET DEFAULT expression
  ```

- usunięcie wartości domyślnej z kolumny:

  ```
  ALTER [COLUMN] column DROP DEFAULT
  ```

- dodanie lub usunięcie ograniczenia `NOT NULL` kolumny (obowiązkowe dane w kolumnie lub nie):

  ```
  ALTER [COLUMN] column {SET|DROP} NOT NULL
  ```

- dodanie lub usunięcie ograniczenia integralnościowego:

  ```
  ADD table_constraint
  DROP CONSTRAINT constraint_name [RESTRICT|CASCADE]
  ```

- zdefiniowanie nowej przestrzeni tabel:

  ```
  SET TABLESPACE new_tablespace
  ```

Zmiana nazwy kolumny:

```
ALTER TABLE [ONLY] name [*]
    RENAME [COLUMN] column TO new_column;
```

Zmiana nazwy tabeli:

```
ALTER TABLE name
    RENAME TO new_name;
```

Zmiana nazwy schematu:

```
ALTER TABLE name
    SET SCHEMA new_schema;
```

Rozważmy dla przykładu tabelę `t_ksiazka` przechowującą informacje o książkach przedstawioną na diagramie ERD na rysunku 2.1. Utworzymy poleceniem `CREATE TABLE` tylko wybrane kolumny tej tabeli tak, aby w przypadku pozostałych zilustrować działanie polecenia `ALTER TABLE`.

```
CREATE TABLE t_ksiazka(
  ISBN int2 PRIMARY KEY,
  tytul varchar(40) NOT NULL,
  wydawca int2,
  dostawca int2,
  cena decimal(4,2),
  ilosc int2,
  CHECK(cena >= 0.0),
  CHECK(ilosc >= 0)
);
```

Zmodyfikujemy początkową strukturę tabeli `t_ksiazka` poleceniami:

- do tabeli `t_ksiazka` dodajemy kolumnę przechowującą rodzaj oprawy książki i ograniczamy wartości w niej przechowywane do tych pochodzących z podanego zbioru:

```
ALTER TABLE t_ksiazka
  ADD oprawa char(6),
  ADD CONSTRAINT t_ksiazka_oprawa_check
  CHECK (oprawa in ('miękka', 'twarda'));
```

- do tabeli `t_ksiazka` dodajemy obowiązkową kolumnę `rok` z wartością domyślną 2024 z ograniczeniem dopasowania do wyrażenia regularnego składającego się z 4 cyfr:

```
ALTER TABLE t_ksiazka
  ADD COLUMN rok char(4),
  ALTER COLUMN rok SET DEFAULT '2024',
  ALTER COLUMN rok SET NOT NULL,
  ADD CONSTRAINT t_ksiazka_rok_check
  CHECK (rok ~ ('^\d{4}$'));
```

- do tabeli `t_ksiazka` dodajemy definicje kluczy obcych pod warunkiem, że tabele `t_wydawca` i `t_dostawca` zostały wcześniej utworzone:

```
ALTER TABLE t_ksiazka
  ADD CONSTRAINT klucz_z_dostawcy
  FOREIGN KEY(dostawca) REFERENCES t_dostawca(id_dostawcy)
  ON DELETE CASCADE;


ALTER TABLE t_ksiazka
  ADD CONSTRAINT klucz_od_wydawcy FOREIGN KEY (wydawca)
  REFERENCES t_wydawca(id_wydawcy)
  ON DELETE CASCADE;
```

2.3. Dodawanie nowych danych do tabeli

W SQL-u można dodawać do tabeli nowe dane na kilka sposobów. Jednym z nich jest bezpośrednie wstawianie wartości do tabeli za pomocą polecenia `INSERT INTO`.

```
INSERT INTO table_name [(column [,...])]
{DEFAULT VALUES | VALUES ({expression|DEFAULT}[,...])[,...] | query}
[RETURNING * | output_expression [[AS] output_name][,...]];
```

Rozważmy przykładową tabelę `t_zamowienie`, której tworzenie pokazaliśmy w podrozdziale 2.1, a która przechowuje podstawowe dane o pojedynczym zamówieniu: identyfikator (`id_zam`), który jest kluczem głównym tabeli, identyfikator klienta (`id_klienta`), który składa zamówienie, datę złożonego zamówienia (domyślnie z wartością bieżącej daty systemowej) (`data_zam`), binarne pole potwierdzające płatność kartą (`karta`) i binarne pole potwierdzające zakończenie realizacji zamówienia (`zrealizowane`). (UWAGA: w przypadku dwóch ostatnich pól binarnych jako typu można użyć `bit(1)`.)

```
CREATE TABLE t_zamowienie (
  id_zam int2 PRIMARY KEY,
  id_klienta int2,
  data_zam DATE DEFAULT current_date,
  karta int2,
  zrealizowane int2,
  CONSTRAINT klucz_od_klienta FOREIGN KEY(id_klienta)
  REFERENCES t_klient(id_klienta) ON DELETE CASCADE
);
```

Wstawianie wartości do wszystkich kolumn w wierszu przyjmie postać:

```
INSERT into t_zamowienie VALUES(1, 1, '2007-01-10', 1, 1);
```

Kolejność podawanych wartości musi w tym przypadku być zgodna z kolejnością kolumn powstałą w chwili tworzenia tabeli. Próba wstawienia wiersza z wartościami w innej kolejności od tej powstałej podczas tworzenia tabeli zakończy się niepowodzeniem i wyświetleniem komunikatu błędu:

```
INSERT into t_zamowienie VALUES('2017-11-21', 1, 1, 2, 3);
ERROR: invalid input syntax for type smallint: "2017-11-21"
LINE 1: INSERT into t_zamowienie VALUES('2017-11-21', 1, 1, 2, 3);
```

Próba wstawienia wiersza z powtarzającą się wartością dla klucza głównego:

```
INSERT into t_zamowienie VALUES(1, 2, '2012-10-13', 1, 1);
```

spowoduje niepowodzenie operacji i pojawienie się komunikatu o błędzie:

```
ERROR: duplicate key value violates unique constraint "t_zamowienie_pkey"
DETAIL: Key (id_zam)=(1) already exists.
```

W poleceniu wstawiania wiersza danych do tabeli możemy jawnie odwołać się do nazw uzupełnianych kolumn, wymieniając je po nazwie tabeli na liście ujętej w nawiasy okrągłe i rozdzielając je przecinkiem, np.:

```
INSERT INTO t_zamowienie (id_zam,id_klienta,data_zam,karta,zrealizowane)
VALUES(123, 3, DEFAULT, 1, 0);
```

Odwołanie się do wartości `DEFAULT` spowoduje wstawienie zdefiniowanej podczas tworzenia tabeli wartości domyślnej dla kolumny `data_zam`.

Dzięki definiowaniu listy kolumn do uzupełnienia, mamy również możliwość zmiany kolejności uzupełnianych kolumn na inną od tej powstałej podczas tworzenia tabeli, np.:

```
INSERT INTO t_zamowienie (id_klienta,data_zam,karta,zrealizowane, id_zam)
VALUES(1, DEFAULT, 0, 0, 124);
```

Ponadto zamiast wielokrotnie wstawiać do tabeli pojedynczy rekord, możemy w jednym poleceniu `INSERT INTO` w klauzuli `VALUES` podać wiele rekordów rozdzielonych przecinkami, np.:

```
INSERT into t_zamowienie VALUES
 (2, 2, '2004-01-10', 1,0),
 (3, 3, '2003-01-10', 0,0),
 (4, 2, '2002-01-11', 0,0),
 (5, 4, '2001-01-11', 1,1),
 (6, 5, '2008-01-11', 1,1),
 (7, 4, '2007-10-12', 1,1),
 (8, 4, '2010-01-12', 1,0);
```

Dzięki definiowaniu listy kolumn, do których mają być wstawiane wartości, możliwe jest również wstawianie wartości dla wybranych kolumn w wierszu. Ważne jest, aby uzupełniane były zawsze kolumny obowiązkowe, w tym kolumna (kolumny) będące kluczem głównym, o ile nie użyto w definicji klucza głównego typu `serial` lub dla kolumn tych nie określono wartości domyślnych. Wartością wstawianą do kolumn, których nie ma na liście i nie zostały dla nich zdefiniowane wartości domyślne, jest `NULL`. Na przykład:

```
INSERT into t_zamowienie (id_zam, id_klienta) VALUES(9, 3);
INSERT into t_zamowienie (id_zam, data_zam) VALUES(10, '2012-01-12');
INSERT into t_zamowienie (id_zam, karta) VALUES(11, 1);
INSERT into t_zamowienie (id_zam) VALUES(12);
```

W każdym z wyżej wymienionych rekordów, tylko data zamówienia (`data_zam`) przyjmie wartość domyślną bieżącej daty systemowej (`current_date`), pozostałe nie wymienione na liście kolumny uzupełnione zostaną wartościami `NULL`. Próba wstawienia rekordu z pominięciem wartości dla klucza głównego zakończy się niepowodzeniem i komunikatem o błędzie:

```
INSERT INTO t_zamowienie(id_klienta) VALUES(3);
ERROR: null value in column "id_zam" violates not-null constraint
DETAIL: Failing row contains (null, 3, 2024-05-02, null, null).
```

W tabeli 2.2 przedstawiamy rekordy tabeli `t_zamowienie` po wykonaniu wszystkich, wymienionych wcześniej, poprawnych instrukcji `INSERT INTO`.

Tabela 2.2: Rekordy tabeli *t_ zamowienie*

id_zam	id_klienta	data_zam	karta	zrealizowane
1	1	2007-01-10	1	1
123	3	2024-05-02	1	0
124	1	2024-05-02	0	0
2	2	2004-01-10	1	0
3	3	2003-01-10	0	0
4	2	2002-01-11	0	0
5	4	2001-01-11	1	1
6	5	2008-01-11	1	1
7	4	2007-10-12	1	1
8	4	2010-01-12	1	0
9	3	2024-05-02		
10		2012-01-12		
11		2024-05-02	1	
12		2024-05-02		

14 rows

2.4. Modyfikowanie i usuwanie danych

Często dane już składowane w bazie danych trzeba zmodyfikować lub usunąć. Do aktualizowania danych znajdujących się w tabeli służy polecenie `UPDATE`, gdy tymczasem polecenie `DELETE` usuwa te dane i, co jest ważne w tym przypadku, usuwane są całe rekordy zawierające te dane.

```
UPDATE table_name [[AS] alias]
SET {column = {expression | DEFAULT} |
               (column [,...]) = ({expression | DEFAULT } [,...])}
    [,...]
[WHERE condition];
```

Modyfikowanie danych tabeli wskazanej nazwą (`table_name`) następuje poprzez zdefiniowanie nowego zbioru danych w klauzuli `SET` dla wskazanych na liście kolumn z opcjonalnym warunkiem, który dane muszą spełniać, aby być zmienione. Brak warunku selekcji danych skutkuje zmodyfikowaniem danych w każdym rekordzie tabeli we wskazanych w klauzuli `SET` kolumnach.

Przykładem zastosowania polecenia `UPDATE` niech będzie zwiększenie ceny każdej książki z wybranych trzech lat o 5%.

```
UPDATE t_ksiazka
SET cena = cena * 1.05
WHERE rok in ('1997','1999', '2012');
```

O poprawności operacji jesteśmy informowani przez PostgreSQL komunikatem o liczbie zmodyfikowanych rekordów: `UPDATE 8`. Aby zobaczyć jak zmodyfikowane zostały dane, należy wykonać na nich polecenie `SELECT`.

Niekiedy musimy usunąć tylko wartość z wiersza. Możemy w tym celu wykorzystać polecenie `UPDATE` przypisując polom w wybranych kolumnach wartość `NULL`. Przykładowo, wiemy, że we wprowadzonym tytule książki o ISBN równym 12 został popełniony błąd. Aby nie wyświetlać błędnych danych do czasu wprowadzenia poprawnego tytułu, zostanie on zmieniony na wartość nieokreśloną `NULL`.

```
UPDATE t_ksiazka
SET tytul = NULL
WHERE ISBN = 12;
```

Gdy chcemy usunąć z tabeli wiersze, możemy użyć w tym celu polecenie `DELETE`.

```
DELETE FROM table_name
[WHERE condition];
```

Na przykład, gdy chcemy usunąć rekordy z zamówieniami sprzed '1990-01-01', wydajemy polecenie:

```
DELETE FROM t_zamowienie
WHERE data_zam < '1990-01-01';
```

Jeśli chcemy usunąć wszystkie rekordy z tabeli, ale nie samą tabelę, możemy zastosować instrukcję `DELETE` bez definiowania warunku.

Aby zilustrować usuwanie wszystkich wierszy z tabeli, stworzymy przykładową tabelę `Example`, wypełnimy ją kilkoma rekordami po to, aby następnie je usunąć.

```
CREATE TABLE Example (one int);
INSERT INTO Example VALUES (1),(2),(3);
DELETE FROM Example;
```

Innym sposobem na usunięcie wszystkich danych bez kasowania tabeli, jest użycie instrukcji `TRUNCATE TABLE`, w której podajemy nazwę tabeli, która ma być wyczyszczona z danych:

```
TRUNCATE TABLE EXAMPLE;
```

2.5. Usuwanie tabeli

Aby usunąć tabelę wraz ze wszystkimi jej wierszami, należy w kliencie `psql` wydać polecenie `DROP TABLE`:

```
DROP TABLE [IF EXISTS] table_name [,...] [CASCADE | RESTRICT];
```

Gdzie `table_name` to nazwa tabeli, która ma być usunięta, a użycie opcji `CASCADE` spowoduje usunięcie wszystkich powiązanych z usuwaną tabelą widoków. W przypadku klucza obcego usunie tylko ograniczenie klucza obcego, a nie całą inną tabelę. Dzięki temu możliwe jest usuwanie w dowolnej kolejności zależnych (pod względem klucza obcego) od siebie tabel. W przypadku wyboru opcji `RESTRICT` nastąpi odmowa usunięcia tabeli, jeśli jakiekolwiek obiekty zależą od niej. Jest to domyślne zachowanie. Jeśli zamierzamy skasować wszystkie dane z tabeli `t_klient` wraz z samą tabelą i wiemy z diagramu 2.1, że powiązana jest ona z tabelą `t_zamowienie`, to jedynie możemy zrobić to poleceniem:

```
DROP TABLE IF EXISTS t_klient CASCADE;
```

Należy pamiętać, że użycie opcji kaskadowego usuwania (`CASCADE`) spowoduje usunięcie w tabeli `t_zamowienie` ograniczenia klucza obcego odnoszącego się do klucza głównego `id_klienta` w usuwanej tabeli oraz wszystkich powiązanych z usuwaną tabelą widoków. Dodatkowo opcja `IF EXISTS` spowoduje, że PostgreSQL najpierw będzie sprawdzał, czy dana tabela istnieje. Jeśli tabeli nie ma w bazie danych, PostgreSQL pomija instrukcję i wyświetla komunikat o braku tabeli, ale nie zgłasza błędu, jak w instrukcji:

```
DROP TABLE t_klient CASCADE;
ERROR:  table "t_klient" does not exist
```

Instrukcja `DROP TABLE IF EXISTS` pozwala ponadto na zautomatyzowanie wykonywania skryptów SQL-owych, w których często używa się jej przed instrukcją `CREATE TABLE`. Jeżeli dana tabela istnieje, dzięki opcji `IF EXISTS` zostanie ona usunięta, zanim ponownie ją utworzymy. Bardzo ważne jest jednak to, aby upewnić się, że automatycznie usuwana tabela rzeczywiście nie jest już używana.

2.6. Zadania do samodzielnego rozwiązania

Utwórz skrypt o nazwie `ksiegarnia.create` zawierający polecenia:

1. Utwórz bazę danych `ksiegarnia`.
2. Połącz się z utworzoną wcześniej bazą danych `ksiegarnia`.
3. Usuń kaskadowo wszystkie tabele przedstawione na diagramie 2.1, o ile istnieją w bazie danych.
4. Utwórz tabele z diagramu 2.1 zgodnie z poniższym opisem:

```
t_ksiazka:
  ISBN int2 klucz główny
  tytul varchar(40) atrybut obowiązkowy
  wydawca int2
  rok char(4) domyślna wartość '2024', atrybut obowiązkowy
    w kolumnie rok dopuszczalna jest tylko wartość
    w formacie 4 cyfr
  oprawa char(6) dopuszczale wartości to 'miękka' lub 'twarda'
  dostawca int2
  cena decimal(4,2) wartość musi być >= 0,0
  ilosc int2 wartość musi być >= 0

t_autor:
  id_autora int2 klucz główny
  imie varchar(40)
  nazwisko varchar(40)

t_a_ksiazki:
  id_autora int2
  ISBN int2

t_wydawca:
  id_wydawcy int2 klucz główny
  nazwa varchar(10)
  adres varchar(40)
```

```
t_zamowienie:
  id_zam int2 klucz główny
  id_klienta int2
  data_zam DATE wartość domyślna current_date
  karta int2
  zrealizowane int2

t_z_ksiazka:
  id_zam int2
  isbn int2
  data_Wys DATE wartość domyślna current_date
  ilosc int2

t_klient:
  id_klienta int2 klucz główny
  imie varchar(30)
  nazwisko varchar(40)
  ulica varchar(30)
  miasto varchar(30),
  wojewodztwo varchar(30)
  kod char(6) ograniczenie dla wartości
  do formatu '00-000' ('0' oznacza dowolną cyfrę)
  telefon char(11) ograniczenie dla wartości
  do formatu '000-000-000' ('0' oznacza dowolną cyfrę)

t_dostawca:
  id_dostawcy int2 klucz główny
  nazwa varchar(30)
  ulica varchar(30)
  miejscowosc varchar(30)
  wojewodztwo varchar(40)
  kod char(6) ograniczenie dla wartości
  do formatu '00-000' ('0' oznacza dowolną cyfrę)
  telefon char(11) ograniczenie dla wartości
  do formatu '000-000-000' ('0' oznacza dowolną cyfrę)
```

5. Zmodyfikuj strukturę tabeli `t_a_ksiazki`, dodając klucz główny złożony z dwóch kolumn `id_autora` i `ISBN` oraz klucze obce dla kolumn `ISBN` będącego referencją do klucza głównego tabeli `t_ksiazka` i `id_autora` będącego referencją do klucza głównego tabeli `t_autor`.

6. Zmodyfikuj strukturę tabeli `t_ksiazka`, dodając klucze obce dla kolumn: `wydawca` będącego referencją do klucza głównego tabeli `t_wydawca` i `dostawca` będącego referencją do klucza głównego tabeli `t_dostawca`.
7. Zmodyfikuj strukturę tabeli `t_zamowienie`, dodając dla kolumny `id_klienta` klucz obcy jako referencję do klucza głównego tabeli `t_klient`.
8. Zmodyfikuj strukturę tabeli `t_z_ksiazka`, dodając klucze obce dla kolumn `id_zam` będącego referencją do klucza głównego tabeli `t_zamowienie` i `ISBN` będącego referencją do klucza głównego tabeli `t_ksiazka`.
9. Wstaw po 10 wierszy do każdej nowo utworzonej tabeli.
10. Korzystając z klienta `psql` uruchom stworzony przez siebie skrypt.

Rozdział 3

Proste zapytania do bazy danych

Język SQL dostarcza instrukcję `SELECT` służącą do pobierania danych z bazy danych. Jest to prawdopodobnie najczęściej używana instrukcja SQL-owa, która składa się z pięciu podstawowych części:

- Operacji – część ta rozpoczyna instrukcję `SELECT` i opisuje operację, która będzie wykonana na danych wskazanych na liście kolumn i funkcji.
- Danych – w klauzuli `FROM`, drugiej w kolejności występowania w poleceniu `SELECT`, wskazywana jest tabela (tabele), z której pobierane są dane wraz z zarezerwowanymi słowami kluczowymi określającymi, jakie dane należy uwzględnić na potrzeby wykonywania obliczeń, filtrowania i pobierania.
- Warunku – występuje w klauzuli `WHERE` i służy do odfiltrowania danych spełniających podany warunek.
- Grupowania – w klauzuli `GROUP BY` podawany jest klucz, względem którego pobierane i łączone są dane, a następnie obliczany jest wynik na podstawie wartości z wszystkich wierszy o tym samym kluczu. Grupowanie może mieć narzucony dodatkowy warunek w klauzuli `HAVING`.
- Przetwarzania końcowego – wynikowe dane są pobierane i przetwarzane, często z użyciem słów kluczowych `ORDER BY` i `LIMIT`.

W tym rozdziale omówimy każdą z wymienionych wyżej części polecenia `SELECT`, ilustrując ich działanie na przykładach oraz podamy ćwiczenia do samodzielnego rozwiązania. Ponieważ wynik zapytania (instrukcji `SELECT`) najczęściej będzie kierowany na standardowe wyjście, czyli ekran, dlatego często będziemy słownie formułowali zapytania w taki sposób, aby wyświetlały pobrane dane na ekranie.

3.1. Ogólna postać polecenia SELECT

Przedstawimy teraz składnię polecenia `SELECT`, ograniczając się do tych jej elementów, które najczęściej występują w zadawanych do bazy danych zapytaniach i które zostaną omówione w tym i następnych rozdziałach. Po szczegółowy opis polecenia `SELECT` zainteresowanego czytelnika odsyłamy do dokumentacji PostgreSQL znajdującej się pod linkiem: `https://www.postgresql.org/docs/15/index.html`.

Ogólna postać instrukcji `SELECT` jest następująca:

```
[WITH [RECURSIVE] with_query [,...]]
SELECT [ALL | DISTINCT [ON (expression[,...])]]
* | expression [[AS] output_name][,...]
[FROM from_item [, ...] ]
[WHERE condition ]
[GROUP BY expression [, ...] ]
[HAVING condition [, ...] ]
[{ UNION | INTERSECT | EXCEPT } [ALL] select ]
[ORDER BY expression [ASC | DESC | USING operator]
[NULLS {FIRST | LAST}] [,...]]
[LIMIT {count | ALL}]
[OFFSET start [ROW | ROWS]];
```

gdzie `from_item` może być jedną z opcji:

```
[ONLY] table_name [*] [[AS] alias [(column_alias[,...])]]
(select) [AS] alias [(column_alias [,...])]
with_query_name [[AS] alias [(column_alias [,...])]]
function_name ([argument [,...]]) [AS] alias
[(column_alias [,...] | column_definition [,...])]
function_name([argument [,...]]) AS (column_definition [,...])
from_item [NATURAL] join_type from_item
[ON join_condition | USING (join_column [,...])];
```

a `query_name` to:

```
with_query_name [(column_name [,...])] AS (select)
TABLE {[ONLY] table_name [*] | with_query_name}
```

Każde zapytanie w wyniku zwraca zbiór elementów – tabelę wirtualną (ang. `Virtual Table`, VT). Czasem jest to zbiór pusty (gdy zapytanie nic nie zwraca), czasem jednoelementowy (może być opisany przez jeden lub wiele atrybutów), ale zazwyczaj zawiera szereg wierszy (rekordów) i kolumn (opisanych przez wiele atrybutów). Kształt tabeli

wynikowej jest określany za pomocą klauzuli `SELECT`. To tutaj określamy, w jaki sposób chcemy definiować elementy zbioru wynikowego (jakimi atrybutami, czyli nazwami kolumn, będą opisane), czy wiersze (rekordy) będą opisane przez wszystkie kolumny z tabel źródłowych (symbol *), czy też tylko przez ich podzbiór.

Najprostsze z możliwych zapytań składa się tylko z klauzuli `SELECT` (nie odwołuje się nawet do żadnej tabeli), w której następuje odwołanie do wartości skalarnej, np. literału, funkcji (np. `current_date`), a zwracany zbiór jest również wartością skalarną – zbiorem jednoelementowym, opisanym jednym atrybutem (jedną kolumną). Dzięki temu możemy na przykład sprawdzić, jaki format daty obowiązuje w systemie PostgreSQL, wydając polecenie:

```
SELECT current_date;
```

i otrzymując jako wynik jeden wiersz opisany jedną kolumną:

```
current_date
--------------
 2024-05-03
(1 row)
```

3.2. Pobieranie danych

Polecenie `SELECT` umożliwia wyszukiwanie informacji w bazie danych przy użyciu operatorów algebry relacji. Zawiera klauzule:

- `SELECT` wskazującą kolumny (atrybuty), których wartości mają być wyświetlone;
- `FROM` wskazującą tabelę (relację), której dotyczy polecenie;
- `WHERE` umożliwiającą wykonanie operacji selekcji z algebry relacji;
- `ORDER BY` umożliwiającą wykonanie operacji sortowania wyników;
- `LIMIT` umożliwiającą pobranie takiej liczby rekordów, która podana jest w tej klauzuli.

3.2.1. Klauzula SELECT

W najogólniejszej postaci, aby pobrać wszystkie dane z tabeli `t_ksiazka`, należy użyć następującego zapytania:

```
SELECT *
FROM t_ksiazka;
```

W klauzuli `SELECT` symbol gwiazdki (`*`) to skrót pozwalający pobrać z bazy wszystkie kolumny z tabeli `t_ksiazka` w kolejności ich tworzenia w poleceniu `CREATE TABLE`. Znak

średnika oznacza koniec instrukcji i jest obowiązkowy w składni języka SQL. (Uwaga: w przypadku klienta `pgAdmin`, średnik dodawany jest automatycznie do każdej wykonywanej instrukcji języka SQL. Niestety może to rodzić problemy, gdy zapomnimy o nim pisząc zapytania, np. w kliencie `psql`.) Rysunek 3.1 przedstawia klika pierwszych wierszy z danych wyjściowych powyższego zapytania.

isbn	tytul	wydawca	rok	oprawa	dostawca	cena	ilosc
1	Kontrabasista	1	1997	twarda	1	20.40	5
2	Mercedes Benc	3	2001	twarda	2	30.00	5
3	Tomik wierszy	2	1998	miękka	3	20.50	4
4	Cesarz laleczka	1	1978	twarda	4	25.50	2
5	Lapidarium	1	1995	twarda	5	35.10	2
6	Pan Tadeusz	4	1997	twarda	6	20.00	5
7	Potop	5	2001	twarda	7	30.00	5
8	Mazurek	6	1998	miękka	1	20.00	4
9	Fortepian Chopina	6	1978	twarda	2	25.30	2

Rysunek 3.1: Fragment wyświetlonych wszystkich danych z tabeli `t_ksiazka`

Jeśli chcemy pobrać zapytaniem tylko określone kolumny, musimy zastąpić gwiazdkę (*) nazwami wybranych kolumn. Kolumny należy podać w kolejności, w jakiej zapytanie ma zwracać dane i rozdzielić je przecinkiem. Przykładowo, jeśli chcemy otrzymać kolumnę `wydawca`, po której następuje kolumna `dostawca` z tabeli `t_ksiazka`, wykonamy zapytanie:

```
SELECT wydawca, dostawca
FROM t_ksiazka;
```

Wynikiem projekcji ograniczonej do wybranego zbioru kolumn (lub pojedynczej kolumny) może być wielokrotne wyświetlenie tych samych informacji (wielokrotne powtarzanie się tych samych rekordów), co pokazano na rysunku 3.2.

wydawca	dostawca
1	1
3	2
2	3
1	4
1	5
4	6
5	7

Rysunek 3.2: Fragment wyświetlonych danych z wybranych kolumn z tabeli `t_ksiazka`

Aby wyeliminować powtarzanie wierszy stosujemy w klauzuli `SELECT` operator `DISTINCT`, np. pobierając z jego pomocą w zapytaniu unikalne wartości kolumny `wydawca` z tabeli `t_ksiazka`. Wyniki poniższego zapytania przedstawione zostały na rysunku 3.3.

```
SELECT DISTINCT wydawca
FROM t_ksiazka;
```

```
 wydawca
---------
       3
       5
       4
       6
       2
       1
(6 rows)
```

Rysunek 3.3: Unikatowe wartości kolumny `wydawca` z tabeli `t_ksiazka`

Klauzula `SELECT` może zawierać dodatkowe elementy służące operacjom na kolumnach. W wyniku ich wykorzystania kolumnom należy nadać nową, alternatywną nazwę – alias, która występuje bezpośrednio po nazwie kolumny. Alias może, ale nie musi, być poprzedzony słowem kluczowym `AS` i może być stosowany dla każdej kolumny występującej w klauzuli `SELECT`, niekoniecznie będącej wyrażeniem.

Ponadto w klauzuli `SELECT` mogą wystąpić dodatkowe elementy takie, jak:

a) **Literały** – dowolny łańcuch znaków, data lub liczba; są dołączane automatycznie do każdego wyświetlanego wiersza, zgodnie z umiejscowieniem w klauzuli `SELECT` oraz do nagłówka kolumny, której dotyczy – w tym przypadku zasadnym jest stosowanie aliasu.
Jako przykład sformułujemy zapytanie, które z tabeli `t_ksiazka` pobierze tytuły książek oraz rok ich wydania poprzedzony literałem `' wydany w roku '` stanowiącym odrębną kolumnę. Wyniki poniższego zapytania przedstawiono na rysunku 3.4.

```
SELECT tytul, ' wydany w roku ' AS "Literał", rok
FROM t_ksiazka;
```

```
          tytul           |     Literał      | rok
--------------------------+------------------+------
 Kontrabasista            |  wydany w roku   | 1997
 Mercedes Benc            |  wydany w roku   | 2001
 Tomik wierszy            |  wydany w roku   | 1998
 Cesarz laleczka          |  wydany w roku   | 1978
 Lapidarium               |  wydany w roku   | 1995
 Pan Tadeusz              |  wydany w roku   | 1997
 Potop                    |  wydany w roku   | 2001
 Mazurek                  |  wydany w roku   | 1998
```

Rysunek 3.4: Użycie literału w zapytaniu

b) **Wyrażenia arytmetyczne** – konstruowane są z nazw kolumn i literałów numerycznych z użyciem operatorów dostępnych w PostgreSQL (patrz podrozdział 1.4). Dzięki zastosowaniu wyrażenia matematycznego możemy na przykład pobrać informacje o tytule książki oraz wartości iloczynu ich ceny (kolumna `cena`) i liczby

egzemplarzy (kolumna `ilosc`) z tabeli `t_ksiazka`. Wyniki poniższego zapytania przedstawiono na rysunku 3.5.

```
SELECT tytul, cena*ilosc "Wyrażenie arytmetyczne"
FROM t_ksiazka;
```

tytul	Wyrażenie arytmetyczne
Kontrabasista	102.00
Mercedes Benc	150.00
Tomik wierszy	82.00
Cesarz laleczka	51.00
Lapidarium	70.20
Pan Tadeusz	100.00
Potop	150.00
Mazurek	80.00

Rysunek 3.5: Użycie wyrażenia arytmetycznego w zapytaniu

c) **Funkcje** – przekształcają wartości kolumn i literałów, do których są stosowane. Szczegółowe informacje o funkcjach PostgreSQL zainteresowany czytelnik odnajdzie w dokumentacji systemu pod linkiem: `https://www.postgresql.org/docs/current/functions.html`
Zastosowanie funkcji zilustrujemy na przykładzie funkcji `COALESCE`, która służy do zamiany wartości `NULL` na dowolną, inną wartość.

Typ wynikowy	Opis	Przykład
`COALESCE(list)`	zwraca pierwszą niepustą wartość z listy (`list`); jeśli wszystkie wartości są `NULL` zwróci `NULL`;	`COALESCE(NULL,1,2)`

W tym celu do tabeli `t_wydawca` dodamy rekord, podając tylko wartość dla klucza głównego. W ten sposób pozostałe kolumny przyjmą wartość `NULL`.

```
INSERT INTO t_wydawca (id_wydawcy)
VALUES (7);
```

Następnie pobieramy nazwy wydawców.

```
SELECT nazwa
FROM t_wydawca;
```

Na rysunku 3.6 pokazano wynik zapytania – wśród siedmiu zwróconych rekordów jest rekord z wartością `NULL`, który jest pusty.
Dzięki zastosowaniu funkcji `COALESCE` możemy w zwracanym wyniku zamienić każdą wartość `NULL` inną wartością. Jest to o tyle istotne, że znak `NULL` nie posiada swojej

```
   nazwa
-----------
 Czytelnik
 PIW
 Znak
 Helion
 Robomatic
 Znak

(7 rows)
```

Rysunek 3.6: `NULL` w wyniku zapytania

interpretacji graficznej i może być w tym przypadku nieopatrznie odczytany jako pusty łańcuch. W wyniku poniższego zapytania dzięki zastosowaniu funkcji `COALESCE` w każdym miejscu wystąpienia wartości `NULL` pojawi się napis `'tu był NULL'`, co ilustruje rysunek 3.7.

```
SELECT COALESCE(nazwa,'tu był NULL') AS nazwa
FROM t_wydawca;
```

```
    nazwa
-------------
 Czytelnik
 PIW
 Znak
 Helion
 Robomatic
 Znak
 tu był NULL
(7 rows)
```

Rysunek 3.7: Funkcja `COALESCE` w wyniku zapytania

Należy pamiętać, że wartość `NULL` „psuje" obliczenia, tzn. wartością wyrażenia, w którym występuje `NULL` będzie `NULL`. Zatem w wyrażeniach matematycznych stosujmy funkcję zastępującą każdą wartość `NULL` wartością 0 lub pomińmy wyrażenia ją zawierające, stosując instrukcję wyboru `CASE`.
Ogólny format instrukcji wyboru ma postać:

```
CASE
  WHEN warunek1 THEN wartość1
  WHEN warunek2 THEN wartość2
  ...
  WHEN warunekN THEN wartośćN
  ELSE wartość
END;
```

Tu `warunek1, warunek2,..., warunekN` są to warunki logiczne, prawdziwość których determinuje odpowiednią wartość. W instrukcji wyboru występuje również

klauzula `ELSE` jako warunek przeciwny do wszystkich opcji występujących wcześniej. Dla każdego rekordu w zapytaniu sprawdzane są warunki w instrukcji `CASE` i po wykryciu pierwszego z nich o wartości `True` instrukcja wyboru zwraca powiązaną z tym warunkiem wartość. Jeśli żaden z warunków nie jest spełniony, zwracana jest wartość powiązana z klauzulą `ELSE`.

Na rysunku 3.8 przedstawiamy wyniki zapytania z użyciem `CASE` wyświetlającego tytuły książek (kolumna `tytul`) oraz bieżącą ich cenę wyliczoną jako różnica ceny (kolumna `cena`) i przyznanego rabatu w zależności od liczby egzemplarzy (kolumna `ilosc`) danej książki.

```
SELECT tytul, cena, (cena - CASE WHEN ilosc IS NULL THEN 0
                                 WHEN ilosc < 5 THEN 10
                                 ELSE -10 END) AS cena_rabat
FROM t_ksiazka;
```

tytul	cena	cena_rabat
Kontrabasista	20.40	30.40
Mercedes Benc	30.00	40.00
Tomik wierszy	20.50	10.50
Cesarz laleczka	25.50	15.50
Lapidarium	35.10	25.10
Pan Tadeusz	20.00	30.00
Potop	30.00	40.00
Mazurek	20.00	10.00

Rysunek 3.8: Wynik zapytania używającego `CASE`

d) **Funkcja konkatenacji** (`||`, `concat()`, `concat_ws(separator, ...)`) - umożliwia łączenie wyświetlanych w poleceniu `SELECT` wartości różnych atrybutów (kolumn) w pojedyncze łańcuchy znaków.

Na rysunkach 3.9, 3.10 oraz 3.11 przedstawiamy wyniki zapytań z użyciem funkcji konkatenacji, odpowiednio dla `||`, `concat` i `concat_ws`, wyświetlające połączone wartości kolumn z tabeli `t_ksiazka`: `ISBN`, `tytul` oraz `rok`.

Używając `||` czy `concat`, należy pamiętać o wstawieniu separatora oddzielającego łączone ze sobą elementy. Z tego względu w prezentowanych niżej przykładach w literałach łańcuchowych uwzględniono spacje.

```
SELECT ISBN || ': ' || tytul || ' wydana w roku: ' ||
       rok AS Konkatenacja
FROM t_ksiazka;

SELECT CONCAT(ISBN, ': ', tytul, ' wydana w roku: ', rok)
FROM t_ksiazka;
```

```
                    konkatenacja
-------------------------------------------------
 1: Kontrabasista wydana w roku: 1997
 2: Mercedes Benc wydana w roku: 2001
 3: Tomik wierszy wydana w roku: 1998
 4: Cesarz laleczka wydana w roku: 1978
 5: Lapidarium wydana w roku: 1995
 6: Pan Tadeusz wydana w roku: 1997
 7: Potop wydana w roku: 2001
 8: Mazurek wydana w roku: 1998
```

Rysunek 3.9: Wynik zapytania używającego ||

```
                       concat
----------------------------------------
 1: Kontrabasista wydana w roku: 1997
 2: Mercedes Benc wydana w roku: 2001
 3: Tomik wierszy wydana w roku: 1998
 4: Cesarz laleczka wydana w roku: 1978
 5: Lapidarium wydana w roku: 1995
 6: Pan Tadeusz wydana w roku: 1997
 7: Potop wydana w roku: 2001
 8: Mazurek wydana w roku: 1998
```

Rysunek 3.10: Wynik zapytania używającego `CONCAT`

W funkcji `concat_ws` jej pierwszy argument jest separatorem, dzięki czemu nie musimy pamiętać o „ręcznym" oddzielaniu łączonych elementów.

```
SELECT CONCAT_WS(' ', ISBN, tytul, 'wydana w roku:', rok)
FROM t_ksiazka;
```

```
                       concat_ws
-------------------------------------
 1 Kontrabasista wydana w roku: 1997
 2 Mercedes Benc wydana w roku: 2001
 3 Tomik wierszy wydana w roku: 1998
 4 Cesarz laleczka wydana w roku: 1978
 5 Lapidarium wydana w roku: 1995
 6 Pan Tadeusz wydana w roku: 1997
 7 Potop wydana w roku: 2001
 8 Mazurek wydana w roku: 1998
```

Rysunek 3.11: Wynik zapytania używającego `CONCAT_WS`

3.2.2. Klauzula WHERE

Klauzula `WHERE` pozwala dokonać selekcji rekordów, które muszą spełniać warunek z tej klauzuli. Warunkiem jest zazwyczaj wyrażenie logiczne, które dla każdego wiersza przyjmuje jedną z wartości: `True` lub `False`. W wyrażeniach logicznych można stosować operatory porównania, operatory logiczne łączące więcej niż jedno wyrażenie logiczne oraz specjalne operatory do obsługi wartości `NULL`.

Między innymi w wyrażeniach logicznych wykorzystywane są:

- Operatory `IN` oraz `NOT IN`, które sprawdzają, czy wartość pobrana z kolumny należy do zbioru jednej lub więcej określonych wartości, np. gdy chcemy zobaczyć tytuły książek, których cena jest jedną z podanych kwot: 10, 20 lub 30, wykonamy zapytanie:

```
SELECT tytul
FROM t_ksiazka
WHERE cena IN (10, 20, 30);
```

Co równoważnie można zapisać, stosując operatory logiczne w następujący sposób:

```
SELECT tytul
FROM t_ksiazka
WHERE cena = 10 OR cena = 20 OR cena = 30;
```

W przypadku stosowania operatorów logicznych w złożonych wyrażeniach logicznych, należy pamiętać o priorytetach operatorów i kolejności ich wykonywania. Jeśli nie jesteśmy pewni, jaki priorytet ma stosowany przez nas operator, należy zastosować nawiasy okrągłe (), które wyznaczą prawidłową kolejność działań.
Przyjrzyjmy się poniższym dwóm zapytaniom, które najlepiej ilustrują problem – zdefiniowania złej kolejności działań, która zmienia znacząco zbiór wyników.

```
SELECT tytul, cena
FROM t_ksiazka
WHERE tytul ilike '%a%' AND cena = 10 OR cena = 20 OR cena = 30;
```

```
     tytul     | cena
---------------+-------
 Mercedes Benc | 30.00
 Pan Tadeusz   | 20.00
 Potop         | 30.00
 Mazurek       | 20.00
 Egzorcysta    | 30.00
(5 rows)
```

Rysunek 3.12: Wynik zapytania ze złą kolejnością działań

Na rysunku 3.12 przedstawiamy zwrócone przez powyższe zapytanie wszystkie rekordy z tabeli `t_ksiazka`, w których tytuł zawiera literę 'a' na dowolnym miejscu oraz których cena wynosi 10 i dodatkowo te rekordy, w których cena wynosi 20 lub 30.
Tymczasem zapytanie z wyznaczoną nawiasami kolejnością działań:

```
SELECT tytul, cena
```

```
FROM t_ksiazka
WHERE tytul ilike '%a%' AND (cena = 10 OR cena = 20 OR cena = 30);
```

zwróci te rekordy z tabeli `t_ksiazka`, których tytuł zawiera literę 'a' na dowolnym miejscu i których cena jest jedną z kwot 10, 20 lub 30 złotych (rysunek 3.13).

```
    tytul     | cena
--------------+-------
 Pan Tadeusz  | 20.00
 Mazurek      | 20.00
 Egzorcysta   | 30.00
(3 rows)
```

Rysunek 3.13: Wynik zapytania z wyznaczoną kolejnością działań

- Operatorów porównania, które w połączeniu z operatorami logicznymi, mogą służyć na przykład do wyboru wartości z podanego przedziału.
  - Cena ma być kwotą z przedziału od 5 do 25 złotych:

    ```
    SELECT tytul, cena
    FROM t_ksiazka
    WHERE cena >= 5 AND cena <= 25;
    ```

    Równoważnie warunek dla przedziału obustronnie domkniętego można zbudować, wykorzystując operator `BETWEEN...AND...`:

    ```
    SELECT tytul, cena
    FROM t_ksiazka
    WHERE cena BETWEEN 5 AND 25;
    ```

  - Cena ma być kwotą spoza przedziału obustronnie domkniętego od 5 do 25 złotych:

    ```
    SELECT tytul, cena
    FROM t_ksiazka
    WHERE cena < 5 OR cena > 25;
    ```

    lub równoważnie z użyciem operatora `NOT`:

    ```
    SELECT tytul, cena
    FROM t_ksiazka
    WHERE NOT (cena >= 5 AND cena <= 25);
    ```

    ```
    SELECT tytul, cena
    FROM t_ksiazka
    WHERE cena NOT BETWEEN 5 AND 25;
    ```

- Porównanie z wzorcem z rozróżnieniem wielkości liter, które odbywa się z użyciem operatorów `LIKE` lub `NOT LIKE` (odpowiednio operatory `ILIKE` lub `NOT ILIKE` ignorują wielkość liter). Sam wzorzec budowany jest między innymi przy użyciu znaku

podkreślenia (’_’) reprezentującego dowolny pojedynczy znak (obowiązkowy) lub znaku procenta (’%’) reprezentującego dowolny (w szczególności pusty) ciąg znaków. Na rysunku 3.14 przedstawiono wyniki zapytania wybierającego z rozróżnieniem wielkości liter imiona i nazwiska tych klientów, którzy mają imiona trzyliterowe i literę ’i’ na końcu nazwiska. Natomiast na rysunku 3.15 przedstawiamy przepisane to samo zapytanie, przy czym w warunku zignorowano wielkość liter.

```
SELECT imie, nazwisko
FROM t_klient
WHERE imie like ’___’ and nazwisko like ’%I’;
```

```
 imie | nazwisko
------+----------
(0 rows)
```

Rysunek 3.14: Wynik zapytania używającego LIKE

```
SELECT imie, nazwisko
FROM t_klient
WHERE imie like ’___’ and nazwisko ilike ’%I’;
```

```
 imie | nazwisko
------+----------
 Jan  | Kowalski
(1 row)
```

Rysunek 3.15: Wynik zapytania używającego ILIKE

3.2.3. Klauzula ORDER BY

Klauzula ORDER BY występuje jako przedostatnia w poleceniu SELECT i powoduje posortowanie wierszy będących wynikiem zapytania według wartości atrybutu(-ów) w niej wskazanego. Jeśli atrybutów sortowania jest więcej, rozdzielamy je przecinkami. Sortowanie wówczas odbywa się według pierwszej kolumny znajdującej się na liście kolumn w klauzuli ORDER BY i dopiero w przypadku tych samych wartości w niej występujących uruchamiane jest sortowanie rekordów wynikowych według kolejnej kolumny. Dla każdej z wymienionych na liście kolumn, względem których sortowane są rekordy, można ustalić odrębny porządek sortowania. Domyślnie sortowanie jest rosnące ASC i aby zmienić porządek sortowania na malejący, należy użyć opcji DESC. Ponieważ kolumny występujące w klauzuli SELECT mają przyporządkowane numery, odpowiednio zaczynając od wartości 1 dla pierwszej kolumny występującej na liście kolumn tej klauzuli i są o jeden zwiększane

dla każdej następnej, w klauzuli `ORDER BY` zamiast nazw kolumn można użyć przyporządkowanych im numerów.

```
SELECT ...
FROM ...
WHERE ...
ORDER BY {nazwa_kolumn(y)|numer_kolumny}[ASC/DESC][,...];
```

Na rysunku 3.16 przedstawiono wyniki zapytania pokazującego niepowtarzające się imiona i nazwiska 5 autorów w porządku rosnącym ich imion, a gdy imiona są takie same w porządku rosnącym ich nazwisk.

```
SELECT DISTINCT imie, nazwisko
FROM t_autor
ORDER BY imie, nazwisko
LIMIT 5;
```

```
  imie   |  nazwisko
---------+------------
 Adam    | Mickiewicz
 Adam    | Zielony
 Beata   | Powstaniec
 Cyprian | Norwid
 Dariusz | Port
(5 rows)
```

Rysunek 3.16: Wynik zapytania ograniczonej liczby pobranych rekordów posortowanych według więcej niż jednej kolumny

Z kolei na rysunku 3.17 przedstawiono wyniki zapytania pokazującego niepowtarzające się imiona i nazwiska 5 autorów posortowane w porządku rosnącym według imion (1), dla tych samych imion uruchamiane jest sortowanie malejące według nazwisk (2 DESC).

```
SELECT DISTINCT imie, nazwisko
FROM t_autor
ORDER BY 1, 2 DESC
LIMIT 5;
```

3.3. Kopiowanie danych

W języku SQL tabela może być również tworzona jako kopia istniejących danych w oparciu o zapytanie `SELECT`. Wówczas polecenie `CREATE TABLE` przyjmuje postać:

```
CREATE TABLE [IF NOT EXISTS] table_name [(col_name[, ...])]
AS QUERY;
```

```
  imie   |  nazwisko
---------+------------
 Adam    | Zielony
 Adam    | Mickiewicz
 Beata   | Powstaniec
 Cyprian | Norwid
 Dariusz | Port
(5 rows)
```

Rysunek 3.17: Wynik zapytania ze złożonym sortowaniem rekordów według kolumn identyfikowanych numerami

W podobny sposób, używając zapytania `SELECT` do danych i uzyskując zbiór wynikowy, możemy wstawić te dane jako wiersze do tabeli, wykorzystując w tym celu polecenie `INSERT INTO` w postaci:

```
INSERT INTO tabel_name [(col_name[,...])]
QUERY;
```

Jako przykład utworzymy tabelę o nazwie `A`, ale tym razem będzie ona kopią już istniejących danych wybranych poleceniem `SELECT`, który zwraca tytuł i cenę książek, o ile cena ta jest różna od 20.

```
CREATE TABLE A AS
SELECT tytul, cena
FROM t_ksiazka
WHERE cena != 20;
```

Następnie poleceniem `\d` klienta `psql` możemy sprawdzić, jaka jest struktura nowo utworzonej tabeli (rysunek 3.18) i wyświetlić jej zawartość (rysunek 3.19):

```
                    Table "public.a"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 tytul  | character varying(40) |           |          |
 cena   | numeric(4,2)          |           |          |
```

Rysunek 3.18: Struktura tabeli `A`

```
SELECT *
FROM A;
```

Następnie dołączamy do tabeli `A` również odrzucone wcześniej dane.

```
INSERT INTO A
SELECT tytul, cena
FROM t_ksiazka
WHERE cena = 20;
```

```
          tytul           | cena
--------------------------+-------
 Kontrabasista            | 20.40
 Mercedes Benc            | 30.00
 Tomik wierszy            | 20.50
 Cesarz laleczka          | 25.50
 Lapidarium               | 35.10
 Potop                    | 30.00
 Fortepian Chopina        | 25.30
 Pole dla wszystkich      | 35.00
    ...
 Madagaskar               | 28.80
 Niebo w ogniu            | 39.99
(43 rows)
```

Rysunek 3.19: Przykładowe rekordy z tabeli `A`

3.4. Zadania do samodzielnego rozwiązania

1. Wyświetlić unikalne wartości kolumny dostawca z tabeli `t_ksiazka`.
2. Wyświetlić ISBN książek oraz wysokość marży, która stanowi 15% ceny książki. Wyniki posortować malejąco według marży.
3. Wyświetlić ISBN oraz wydawców książek dostarczanych przez dostawcę o identyfikatorze 2. Wyniki posortować rosnąco według kolumny ISBN.
4. Wyświetlić ISBN, cenę oraz ilość książek, o ile ilość nie przekracza 5 egzemplarzy.
5. Wyświetlić ISBN, tytuł oraz cenę tych książek, których ilość egzemplarzy znajduje się w przedziale od 1 do 4.
6. Wyświetlić nazwiska i imiona tych klientów, których identyfikator równy jest 1 lub 3.
7. Wyświetlić nazwisko tych klientów, których imię zawiera literę 'a' na dowolnym miejscu w łańcuchu znaków. Wyniki uporządkować rosnąco.
8. Wyświetlić nazwisko oraz dane adresowe tych klientów, którzy mają na imię 'Adam' i ich identyfikator jest nie większy niż 5.
9. Wyświetlić ISBN tych książek wydanych przez 'PWN' lub 'HELION', których cena jest z zakresu od 8 do 30 i nie mają oprawy miękkiej.
10. Wyświetlić w porządku malejącym nazwiska tych klientów, których imię zaczyna i kończy się na literę `'a'`.
11. Wyświetlić tytuły i ceny książek. Ponadto, jeśli cena:
    - `< 30`, wyświetlić tekst: `'poniżej trzydziestu zł'`,
    - `= 30` – wyświetlić tekst: `'równo trzydzieści złotych'`,
    - `> 30`, wyświetlić tekst: `'powyżej trzydziestu złotych'`.

    Wyniki uporządkować rosnąco według tytułu książki i ograniczyć wyniki do 10 rekordów.

12. Wyświetlić informacje o ISBN, tytule i roku tych książek, które nie są wydane ani w roku 1997, ani w 1998.
13. Wyświetlić te rekordy z tabeli `t_wydawca`, dla których nazwa lub adres nie zostały podane przy wstawianiu danych do tabeli.
14. Wyświetlić wszystkie informacje o książkach i posortować w pierwszej kolejności rosnąco według wydawcy potem malejąco według dostawcy.
15. Wyświetlić dla każdej książki tytuł i obliczony iloczyn jej ceny i ilości (nadać kolumnie alias STAN). Wiersze uporządkować w malejącej kolejności stanu, książki z takim samym stanem uporządkować w rosnącej kolejności tytułów.
16. Wyświetlić identyfikator i nazwę wydawców mających siedzibę na ulicy 'Woronicza', wyniki uporządkować malejąco według identyfikatorów.
17. Wyświetlić informacje na temat zamówień (`id_zam`, `data_zam`), które zostały złożone między 20 listopada, a 31 grudnia 2015 roku.
18. Wyświetlić informacje na temat klientów (`id_klienta`, `nazwisko`), którzy posiadają dwuczłonowe nazwisko. Można założyć, że nazwisko dwuczłonowe jest rozdzielone łącznikiem (znak `'-'`, można użyć np. funkcji `POSITION`).
19. Wyświetlić ISBN i tytuły książek, których ilość egzemplarzy jest zerowa.
20. Wyświetlić wszystkie informacje o klientach, którzy mieszkają w województwie mazowieckim lub których imię i nazwisko są łańcuchami tej samej długości. (`LENGTH`)
21. Wyświetlić dane o klientach, których nazwiska zaczynają się na literę `'M'`. Wyświetlany tekst ma zostać uzupełniony do 20 znaków sekwencją znaków `'#*.'` Należy użyć funkcji `RPAD`. Czym różni się w działaniu funkcja `LPAD` od `RPAD`?
22. Wyświetlić w pierwszej kolumnie nazwiska klientów. W drugiej kolumnie wyświetlić tylko 3 znaki z nazwiska poczynając od 1 znaku. Użyć funkcji `SUBSTR`.
23. Wyświetlić dane o autorach, których nazwiska są dłuższe niż 6 znaków oraz w osobnej kolumnie dla potwierdzenia ich długość. Wynik posortować wg. długości nazwiska w porządku od największej do najmniejszej.
24. Wyświetlić w pierwszej kolumnie nazwiska klientów, a w drugiej kolumnie nazwiska, ale każdą literę `'a'` zastąpić przez `'X'`. Użyć funkcji `REPLACE`.
25. Wyświetlić w pierwszej kolumnie nazwiska autorów, a w drugiej kolumnie nazwiska, ale pisane wspak. Użyć funkcji `REVERSE`.
26. Wyświetlić wszystkie informacje o autorach, których nazwiska zaczynają się na literę `'K'`, przy czym w nazwisku wszystkie litery mają być małe, a w imieniu duże. Użyć funkcji `LOWER` oraz `UPPER`.
27. Wyświetlić w pierwszej kolumnie nazwiska autorów. W drugiej kolumnie wypisać, w którym miejscu po raz pierwszy pojawiła się litera `'a'`. Użyć funkcji `POSITION`.
28. Wyświetlić ISBN i zaokrągloną do wartości całkowitej cenę książek. Użyć funkcji `ROUND`.

29. Wyświetlić datę zamówienia w domyślnym formacie wyświetlania oraz w formacie ustalonym samodzielnie. Użyć funkcji `TO_CHAR`.
30. Wyświetlić ostateczną datę wysyłki zamówienia, która nie może przekroczyć 7 dni od złożenia zamówienia. Użyć np. funkcji `INTERVAL`.

Rozdział 4

Łączenie danych

Wyszukiwane dane najczęściej znajdują się w kilku tabelach. Wówczas do ich pobrania nie wystarcza prosta instrukcja `SELECT` odnosząca się do pojedynczej tabeli. Język SQL dostarcza metod łączenia poziomego powiązanych danych pochodzących z różnych tabel oraz metod łączenia pionowego wyników kilku zapytań (co najmniej dwóch), które omówimy w tym rozdziale.

4.1. Łączenie poziome tabel

Tabele łączymy na podstawie wartości wspólnego atrybutu, na przykład wartości pary klucz podstawowy – klucz obcy, używając ich nazw. Często są one takie same i aby uniknąć błędów kompilacji, nazwę kolumny poprzedzamy nazwą tabeli, z której pochodzi, rozdzielając je kropką, np. `t_ksiazka.ISBN`. Ze względu na długość nazw tabel, kolumn, aby poprawić czytelność zapytania, stosujemy aliasy.

Tabele łączymy zgodnie z następującymi wskazówkami:

1. Staramy się łączyć tabele za pomocą kolumn przechowujących parę kluczy podstawowy-klucz obcy.
2. Do złączenia używamy całych kluczy podstawowych tabel. Jeżeli dla jakiejś tabeli zdefiniowano złożony (składający się z kilku atrybutów) klucz podstawowy, klucz obcy również musi być złożony i odwołujemy się do całych takich kluczy.
3. Łączymy obiekty za pomocą kolumn tego samego typu, pochodzących z tej samej dziedziny (przez dziedzinę rozumiemy zbiór danych o tym samym znaczeniu, np. dziedziną dla kolumny `imie` będzie zbiór imion).
4. Poprzedzamy nazwy kolumn nazwą obiektu źródłowego, nawet jeżeli nazwy te są unikatowe – w ten sposób poprawimy czytelność zapytania.
5. Ograniczamy liczbę łączonych obiektów do niezbędnego minimum.

Niezależnie od wybranego typu złączenia, w wyniku przetwarzania klauzuli `FROM` otrzymujemy zawsze zbiór elementów – tabelę wirtualną, opisany za pomocą wszystkich kolumn łączonych tabel. W tym momencie nie ma znaczenia, czy łączymy dwie czy więcej tabel połączeniem wewnętrznym lub zewnętrznym. Elementy (rekordy, wiersze) tabeli wynikowej, będą określone zawsze przez wszystkie atrybuty (kolumny) łączonych zbiorów. W klauzuli `SELECT` możemy zawęzić wynikowy zbiór atrybutów do tych nas interesujących. Po wybraniu tabel musimy określić typ złączenia oraz jego warunki. Tutaj stosujemy w praktyce wiedzę na temat relacyjnych baz danych i sposobów powiązań tabel między sobą. Należy pamiętać, że niezależnie od liczby tabel (trzech i więcej), łączenie ich sprowadza się zawsze do wielokrotnego wykonania operacji łączenia dwóch z nich.

Ponieważ teoria baz danych nie jest celem tego podręcznika (zakładamy, że czytelnik ją już posiada), chcielibyśmy tylko zwrócić uwagę i zainteresować czytelnika faktem, że zdefiniowane w obrębie bazy danych `ksiegarnia` klucze obce, będące ograniczeniami dla kolumn w tabelach, zwiększają integralność danych poprzez zagwarantowanie, że klucz obcy nigdy nie będzie zawierał wartości nieobecnej we wskazanej tabeli. Jest to tak zwana `integralność referencyjna`. Ponadto dodanie ograniczenia klucza obcego pomaga dodatkowo zwiększyć wydajność bazy danych. Natomiast w większości analitycznych baz danych klucze obce nie są używane.

4.1.1. Złączenie wewnętrzne

Złączenie wewnętrzne (`INNER JOIN`) łączy ze sobą różne tabele na podstawie warunku nazywanego predykatem. W wyniku połączenia otrzymujemy tabelę wynikową, składającą się ze wszystkich kolumn tabel wejściowych zawierającą tylko te rekordy, dla których warunki złączenia wewnętrznego będą spełnione (w logice trójwartościowej, wynik musi być `True`). W przeciwnym razie dana kombinacja jest pomijana w wynikach.

Złączenia wewnętrzne są zwykle zapisywane w następujący sposób:

```
SELECT column [,...]
FROM T1 [INNER] JOIN T2 {ON bool_exp | USING(join_column_list)}
     [[INNER] JOIN ...];
```

`INNER JOIN` jest złączeniem symetrycznym i nie ma specjalnego znaczenia, czy łączymy tabelę `T1` z `T2` czy odwrotnie. Podobnie z warunkami w `ON`. Dla porządku sensownie jest jednak zachować kolejność atrybutów w `ON` po tej samej stronie co określenie tabel źródłowych wobec operatora `JOIN`. Oczywiście w jednym zapytaniu możemy połączyć więcej niż dwie tabele, co zostanie zilustrowane w ostatnim przykładzie tego podrozdziału.

Rozważmy teraz przykład pobrania danych pochodzących z różnych tabel. W wyniku połączenia tabel `t_ksiazka` z `t_wydawca` do tabeli wynikowej ograniczonej do dwóch kolumn `t_wydawca.nazwa` i `t_ksiazka.tytul` zostały dołączone tylko te dane, dla których

wartości kolumny `t_ksiazka.wydawca` oraz kolumny `t_wydawca.id_wydawcy` są takie same. Rysunek 4.1 przedstawia pięć pierwszych wierszy z danych wyjściowych.

```
SELECT t_wydawca.nazwa, t_ksiazka.tytul
FROM t_ksiazka INNER JOIN t_wydawca
     ON t_ksiazka.wydawca = t_wydawca.id_wydawcy
LIMIT 5;
```

```
   nazwa   |      tytul
-----------+----------------
 Czytelnik | Kontrabasista
 Znak      | Mercedes Benc
 PIW       | Tomik wierszy
 Czytelnik | Cesarz laleczka
 Czytelnik | Lapidarium
(5 rows)
```

Rysunek 4.1: Tabela `t_ksiazka` złączona z tabelą `t_wydawca`

Tabelom występującym w złączeniu można nadać aliasy, aby nie pisać za każdym razem całych, często długich ich nazw.

```
SELECT nazwisko, ISBN
FROM t_autor ta INNER JOIN t_a_ksiazki AS tak
ON ta.id_autora = tak.id_autora
LIMIT 5;
```

Odwoływanie się do nazw kolumn z odniesieniem do nazwy tabeli, w której się znajdują, jest istotne w momencie, gdy są one takie same. PostgreSQL musi otrzymać jednoznaczną informację, z którą z kolumn ma do czynienia.

```
SELECT id_autora, nazwisko, ISBN
FROM t_autor ta INNER JOIN t_a_ksiazki AS tak
ON ta.id_autora = tak.id_autora
LIMIT 5;
```

W tym przypadku użycie w złączeniu `INNER JOIN` frazy `ON` spowoduje wystąpienie błędu i pojawienie się na ekranie komunikatu:

```
ERROR:  column reference "id_autora" is ambiguous
LINE 1: SELECT id_autora, nazwisko, ISBN
```

i potrzebę zmiany zapytania poprzez wskazanie nazwy tabeli w klauzuli `SELECT`, z której kolumna `id_autora` pochodzi.

W przypadku zgodności nazw kolumn występujących w warunku złączenia, `INNER JOIN` dostarcza składni z użyciem frazy `USING`, dla której w nawiasie okrągłym podaje się

nazwę kolumny łącznikowej. Tu powtarzająca się kolumna jest usuwana z tabeli wynikowej, a zapytanie pobiera dane i nie wskazuje żadnych błędów. Na rysunku 4.2 pokazano 5 rekordów tabeli wynikowej uzyskanej z użyciem frazy `USING`.

```
SELECT id_autora, nazwisko, ISBN
FROM t_autor INNER JOIN t_a_ksiazki USING (id_autora)
LIMIT 5;
```

```
 id_autora | nazwisko | isbn
-----------+----------+------
         2 | Suskind  |    1
         2 | Suskind  |    5
         2 | Suskind  |   13
         2 | Suskind  |   31
         3 | Kapusta  |    4
(5 rows)
```

Rysunek 4.2: Złączenie z użyciem słowa kluczowego `USING`

Wszystkie do tej pory prezentowane przykłady łączyły tabele po kolumnach będących jednocześnie kluczami obcymi/podstawowymi tabel. Ogólną zasadą łączenia tabel jest możliwość jego realizacji po dowolnych kolumnach. Musi być spełniony tylko jeden warunek – zgodność typów danych łączonych atrybutów. To, jak zapiszemy warunek i czy będzie miał sens, zależy tylko od nas – język SQL nie wprowadza tu żadnych ograniczeń. Dodatkowo na wartościach atrybutów, po których łączymy, możemy wykonywać dowolne operacje: przetwarzać je za pomocą funkcji skalarnych, wykonywać działania arytmetyczne, łączenia stringów itp..

PostgreSQL wprowadza jeszcze jeden sposób złączenia wewnętrznego opartego właśnie na samodzielnym doborze przez system kolumn, względem których odbywa się łączenie tabel. Jest to `NATURAL JOIN`, który nie wymaga odniesienia i warunku dla kolumn łącznikowych, a podczas jego wykonania poszukiwane są w tabelach kolumny o tej samej nazwie i zgodności typów. Na rysunku 4.3 przedstawiono 5 rekordów tabeli wynikowej złączenia naturalnego.

```
SELECT id_autora, nazwisko, ISBN
FROM t_autor NATURAL JOIN t_a_ksiazki
LIMIT 5;
```

W przypadku stosowania złączenia wewnętrznego `NATURAL JOIN` trzeba być pewnym poprawności wyboru kolumn łączenia. Ze względu na możliwość wystąpienia więcej niż jednej kolumny o tej samej nazwie i tym samym typie danych w niej składowanych złączenie naturalne może prowadzić do niepoprawnego połączenia tabel (niezgodnego z diagramem ERD) i pobrać niepoprawne dane. Zatem to po stronie projektanta i użytkownika

```
 id_autora | nazwisko | isbn
-----------+----------+------
         2 | Suskind  |    1
         2 | Suskind  |    5
         2 | Suskind  |   13
         2 | Suskind  |   31
         3 | Kapusta  |    4
(5 rows)
```

Rysunek 4.3: Złączenie `NATURAL JOIN`

baz danych leży obowiązek zadbania o poprawne odzwierciedlenie danych rzeczywistych i występujących zależności pomiędzy nimi (integralność bazy danych).

Złączenia wewnętrzne `INNER JOIN` są zgodne ze standardem `ANSI X3.135` i powinny być stosowane w produkcyjnych bazach danych. Istnieje jednak jeszcze inna, starsza metoda zapisu łączenia tabel, która już od standardu `ANSI SQL:92` nie jest zalecana, wręcz mówi się po prostu o jej nie stosowaniu. Jest to złączenie wewnętrzne, w którym definiowanie warunku zamiast w `ON/USING` odbywa się w klauzuli `WHERE`. Łączone iloczynem kartezjańskim tabele wymienia się kolejno w klauzuli `FROM`, oddzielając je przecinkami. Otrzymana w wyniku tej operacji tabela wynikowa zawiera wszystkie kolumny z każdej wymienionej na liście tabeli i wypełniona jest rekordami będącymi połączeniem rekordu jednej tabeli ze wszystkimi rekordami drugiej tabeli itd.. Następnie sprawdzany jest dla każdego rekordu tabeli wynikowej warunek złączenia w klauzuli `WHERE` i odrzucane są rekordy nie spełniające tego warunku.

Na rysunku 4.4 przedstawiamy pięć pierwszych rekordów tabeli wynikowej zapytania zwracającego wszystkie informacje o autorze i tytuł napisanej przez niego książki. W przypadku łączenia tabel odwołanie `ta.*` pozwala na uwzględnienie w tabeli wynikowej wszystkich kolumn pochodzących z tabeli oznaczonej aliasem `ta`, czyli tabeli `t_autor`.

```
SELECT ta.*, tytul
FROM t_autor ta, t_a_ksiazki tak, t_ksiazka tk
WHERE ta.id_autora = tak.id_autora AND tak.ISBN = tk.ISBN
LIMIT 5;
```

```
 id_autora |  imie   | nazwisko |      tytul
-----------+---------+----------+-----------------
         2 | Patrick | Suskind  | Kontrabasista
         2 | Patrick | Suskind  | Lapidarium
         2 | Patrick | Suskind  | Zimna woda
         2 | Patrick | Suskind  | M jak masakra
         3 | Ryszard | Kapusta  | Cesarz laleczka
(5 rows)
```

Rysunek 4.4: Złączenie tabel iloczynem kartezjańskim z predykatem w klauzuli `WHERE`

Przepisane powyższe zapytanie z użyciem złączenia wewnętrznego ma postać:

```
SELECT ta.*, tytul
FROM t_autor ta INNER JOIN  t_a_ksiazki tak USING(id_autora)
     INNER JOIN t_ksiazka tk ON tak.ISBN = tk.ISBN
LIMIT 5;
```

Na rysunku 4.5 przedstawiamy wybrane kolumny tabeli wynikowej, utworzonej przez połączenie wszystkich tabel bazy danych `ksiegarnia`. Dane zostały ograniczone do książek wydanych w określonym roku oraz klientów, których nazwisko kończy się na `'ski'`.

```
SELECT t_klient.nazwisko AS Klient, ISBN, t_autor.nazwisko AS Autor,
       t_dostawca.nazwa AS Dostawca, t_wydawca.nazwa AS Wydawca
FROM t_klient JOIN t_zamowienie USING(id_klienta)
     JOIN t_z_ksiazka USING (id_zam)
     JOIN t_ksiazka USING (ISBN)
     JOIN t_a_ksiazki USING (ISBN)
     JOIN t_autor USING (id_autora)
     JOIN t_dostawca ON dostawca = id_dostawcy
     JOIN t_wydawca ON wydawca = id_wydawcy
WHERE rok in ('1997','2002','1998')
      AND t_klient.nazwisko ILIKE '%ski';
```

```
   klient   | isbn |  autor  | dostawca |  wydawca
------------+------+---------+----------+-----------
 Kowalski   |    1 | Suskind | Goniec   | Czytelnik
 Kowalski   |    3 | Kundera | Konik    | PIW
 Malinowski |    3 | Kundera | Konik    | PIW
 Kowalski   |    8 | Wybicki | Goniec   | Znak
(4 rows)
```

Rysunek 4.5: Wybrane dane z połączonych tabel bazy `ksiegarnia`

4.1.2. Złączenie zewnętrzne

W złączeniu zewnętrznym wiersze nie spełniające warunku połączenia mogą być wyświetlone w wyniku zapytania. Operator połączenia zewnętrznego stosujemy, gdy po jednej stronie w tabeli brakuje rekordów umożliwiających wyświetlenie wszystkich rekordów drugiej tabeli (po połączeniu). Złączenie zewnętrzne tworzy dodatkowe puste rekordy w niepełnej tabeli. Liczba pustych rekordów jest równa liczbie rekordów, które nie spełniają warunków klasycznego połączenia.

Zanim przejdziemy do omówienia kategorii złączeń zewnętrznych, aby zilustrować ich działanie do tabeli `t_wydawca` dodamy dwa rekordy:

```
INSERT INTO t_wydawca
VALUES(8, 'PWN','02-460 Warszawa, ul. Daimlera 2');
INSERT INTO t_wydawca
VALUES(9, 'Novae Res','81-368 Gdynia, ul. Nowa 9/4');
```

Złączenia zewnętrzne można podzielić na trzy kategorie:

- `Złączenie zewnętrzne lewostronne` – zwracane są wszystkie wiersze tabeli podanej po lewej stronie klauzuli `JOIN`. Jeśli instrukcja nie znajdzie pasującego wiersza z drugiej tabeli (podanej po prawej stronie operatora), zwraca wiersz `NULL`. Złączenia tego typu tworzone są za pomocą słów kluczowych `LEFT OUTER JOIN`, po których należy podać predykat złączenia. Można używać również skróconego zapisu `LEFT JOIN`.
  Na rysunku 4.6 pokazujemy tabelę wynikową lewostronnego złączenia zewnętrznego tabel `t_wydawca` (z lewej strony operatora `LEFT JOIN`) z tabelą `t_ksiazka`, w której rekordy ograniczone są tylko do tych, do których nie ma odwołania w tabeli `t_ksiazka` i generowany jest dla nich rekord `NULL`.

  ```
  SELECT tytul, nazwa
  FROM t_wydawca LEFT OUTER JOIN t_ksiazka
  ON (id_wydawcy = wydawca)
  WHERE tytul is NULL;
  ```

  ```
   tytul |   nazwa
  -------+-----------
         | PWN
         | Novae Res
         |
  (3 rows)
  ```

  Rysunek 4.6: Złączenie zewnętrzne lewostronne

- `Złączenie zewnętrzne prawostronne` – zwracane są wszystkie wiersze tabeli podanej po prawej stronie klauzuli `JOIN`. Jeśli instrukcja nie znajdzie pasującego wiersza z drugiej tabeli (podanej po lewej stronie), zwraca wiersz `NULL`. Złączenia tego typu tworzone są za pomocą słów kluczowych `RIGHT OUTER JOIN`, po których należy podać predykat złączenia. Można używać również skróconego zapisu `RIGHT JOIN`.
  Na rysunku 4.7 pokazujemy tabelę wynikową prawostronnego złączenia zewnętrznego tabel `t_wydawca` (z prawej strony operatora `RIGHT JOIN`) z tabelą `t_ksiazka`, w której rekordy ograniczone są tylko do tych, do których nie ma odwołania w tabeli `t_ksiazka` i generowany jest dla nich rekord `NULL`. Zmiana kolejności tabel w złączeniu prawostronnym, symetrycznie do lewostronnego złączenia, spowodowała wygenerowanie tej samej tabeli wynikowej.

```
SELECT tytul, nazwa
FROM t_ksiazka RIGHT OUTER JOIN t_wydawca
ON (id_wydawcy = wydawca)
WHERE tytul is NULL;
```

```
 tytul |   nazwa
-------+-----------
       | PWN
       | Novae Res
       |
(3 rows)
```

Rysunek 4.7: Złączenie zewnętrzne prawostronne

- `Pełne złączenie zewnętrzne` – zwraca wszystkie wiersze z lewej i prawej tabeli niezależnie od tego, czy predykat złączenia jest spełniony. W przypadku rekordów, dla których predykat jest spełniony, rekordy z obu tabel są odpowiednio łączone. Rekordy, dla których predykat nie jest spełniony, są zapełniane wartością `NULL`. Pełne złączenie zewnętrzne jest wykonywane za pomocą klauzuli `FULL OUTER JOIN`, po której należy podać predykat złączenia. Jeszcze raz powtarzamy złączenie tabel `t_ksiazka` i `t_wydawca`, tym razem w pełnym złączeniu zewnętrznym (rysunek 4.8).

```
SELECT tytul, nazwa
FROM t_ksiazka FULL OUTER JOIN t_wydawca
ON (id_wydawcy = wydawca)
WHERE tytul is NULL;
```

```
 tytul |   nazwa
-------+-----------
       | PWN
       | Novae Res
       |
(3 rows)
```

Rysunek 4.8: Pełne złączenie zewnętrzne

4.1.3. Iloczyn kartezjański - `CROSS JOIN`

W języku SQL istnieje złączenie krzyżowe `CROSS JOIN`, które jest iloczynem kartezjańskim bez warunków - rzadko stosowane. Łączy każdy wiersz tabeli A z każdym wierszem tabeli B. Jako jedyne nie ma możliwości utworzenia warunków połączenia w `ON|USING`, bo z założenia ma połączyć wszystko ze wszystkim.

Jako przykład rozpatrzmy potrzebę przeprowadzenia analizy koszykowej dotyczącej wzorców sprzedażowych z uwzględnieniem wielu książek. Chodzi o sytuację, w której łączenie dwóch tytułów i ich sprzedaż po cenie promocyjnej przynosi korzyść księgarni. W celu analizy koszykowej należy ustalić za pomocą złączenia krzyżowego wszystkie kombinacje dwóch książek, jakie możemy utworzyć na podstawie zbioru danych z tabeli `t_ksiazka`. W przykładzie ograniczymy się tylko do 5 rekordów tabeli wynikowej przedstawionych na rysunku 4.9.

```
SELECT tk1.ISBN, tk1.tytul, tk2.ISBN, tk2.tytul
FROM t_ksiazka tk1 CROSS JOIN t_ksiazka tk2
LIMIT 5;
```

```
 isbn |     tytul     | isbn |      tytul
------+---------------+------+-----------------
    1 | Kontrabasista |    1 | Kontrabasista
    1 | Kontrabasista |    2 | Mercedes Benc
    1 | Kontrabasista |    3 | Tomik wierszy
    1 | Kontrabasista |    4 | Cesarz laleczka
    1 | Kontrabasista |    5 | Lapidarium
(5 rows)
```

Rysunek 4.9: Przykład złączenia krzyżowego

Powyższa instrukcja, bez ograniczenia `LIMIT`, zwraca wszystkie możliwe kombinacje wskazanych wartości z tej samej tabeli. Wyniki zapytania (zgodnie z liczbą wierszy w tabeli `t_ksiazka` przykładowej bazy danych `ksiegarnia` – 44 wiersze) obejmują 1936 rekordów. W praktyce złączenia krzyżowe nie są stosowane, ze względu na niebezpieczeństwo zablokowania bazy danych i doprowadzenia do jej awarii w wyniku powstania wielkiej (setek miliardów!) liczby rekordów. Stosowanie złączenia krzyżowego pozostawiamy pod rozwagę użytkownikowi systemu bazodanowego, licząc na jego daleko posuniętą ostrożność.

Złączenie typu `CROSS JOIN` jest realizowane również wtedy, gdy wyszczególnimy tabele w klauzuli `FROM`, separując je tylko przecinkiem.

```
SELECT *
FROM t_wydawca, t_dostawca;
```

4.2. Łączenie pionowe relacji

W celu pionowego połączenia relacji, czyli połączenia wyników dwóch lub więcej zapytań, stosujemy jeden z operatorów zbiorowych:

```
<SELECT . . . > UNION [ALL] <SELECT . . . >
<SELECT . . . > INTERSECT [ALL] <SELECT . . . >
<SELECT . . . > EXCEPT [ALL] <SELECT . . . >
```

Operatory te działają na wynikach co najmniej dwóch operacji SELECT. W łączonych operatorem zbiorowym klauzulach SELECT musi wystąpić ta sama liczba kolumn o typach zgodnych i zgodnych dziedzinach. W wyniku zapytania pojawią się nazwy atrybutów wyłącznie z pierwszej klauzuli SELECT. Polecenia SELECT wykonywane są w kolejności ich wystąpienia (od góry do dołu). W celu zmiany tej kolejności stosujemy nawiasy. Jeśli wystąpi potrzeba użycia klauzuli ORDER BY, to musi ona wystąpić jako ostatnia klauzula zapytania. Ponadto, w klauzuli ORDER BY nie stosujemy nazw kolumn, lecz ich numery porządkowe.

```
<SELECT ...> OPERATOR <SELECT ...> <ORDER BY ...>;
```

W poniższym przykładzie łączymy w jedną tabelę wyniki dwóch zapytań: pierwszego identyfikującego tytuły i wydawców książek dostawcy o identyfikatorze 1 i drugiego ograniczonego do tych samych kolumn (ważne, aby w obydwu zapytaniach kolumny w klauzuli SELECT wymienione były w tej samej kolejności), ale dla dostawcy o identyfikatorze 3. Dane porządkujemy malejąco według tytułu książek.

```
SELECT tytul, wydawca FROM t_ksiazka WHERE dostawca = 1
UNION ALL
SELECT tytul, wydawca FROM t_ksiazka WHERE dostawca = 3
ORDER BY 1 DESC;
```

4.3. Zadania do samodzielnego rozwiązania

1. Wyświetlić wszystkie informacje o tych autorach, których cena książki jest z przedziału 10 do 15. Wiersze uporządkować malejąco według nazwiska autora.
2. Wyświetlić dla każdej książki tytuł, nazwę jej wydawcy oraz nazwę jej dostawcy. Wiersze uporządkować w malejącej kolejności tytułów.
3. Wyświetlić ISBN, tytuł, nazwę dostawcy oraz nazwisko klienta tych książek, które napisał autor o imieniu Adam.
4. Wyświetlić nazwę wydawcy, który wydał książki o ISBN z przedziału ⟨1,10⟩.
5. Wyświetlić tytuły książek, dla których zamówienie zostało już zrealizowane.
6. Wyświetlić nazwy tych wydawców, których książek nasza księgarnia nie ma na stanie.
7. Wyświetlić ISBN książek, imię i nazwisko autora oraz imię i nazwisko tych klientów, którzy zamówili więcej niż jedną książkę.
8. Wyświetlić dane o klientach, którzy wybrali książki w miękkiej oprawie, a ich zamówienie zostało złożone przed 1 stycznia 2001 roku.
9. Wyświetlić informację o tych dostawcach, których siedziba jest na tej samej ulicy, co ulica na której mieszka klient.

10. Wyświetlić `id_klienta`, imię i nazwisko tych klientów, którzy zamówili książki wydane w roku 2012.
11. Wyświetlić ISBN, tytuł, imię i nazwisko autora tych książek, które zostały wydane przez 'PIW', 'Znak' lub 'Helion'.
12. Wypisać ISBN, tytuł, imię i nazwisko autora tych książek, które nie zostały dostarczone przez 'UPS' lub 'Konik'.
13. Wyświetlić tytuły i ceny książek. Ponadto, jeśli cena jest mniejsza od 30, wyświetlić tekst: 'poniżej 30', jeśli cena jest równa 30 – wyświetlić tekst: 'równo 30', natomiast jeśli cena jest większa od 30, wyświetlić tekst: 'powyżej 30'. Uporządkować wiersze rosnąco według ceny. Użyć odpowiedniego operatora zbiorowego.
14. Wyświetlić nazwy tych wydawców, których książek nie ma jeszcze w księgarni. Użyć odpowiedniego operatora zbiorowego.
15. Wyświetlić wszystkie informacje o książkach, których cena mieści się w przedziale od `11.50` do `33.50` lub w numerze telefonu klienta, który je kupił na dowolnej pozycji znajduje się znak cyfry 3 lub znak cyfry 4.
16. Wyświetlić dla każdego klienta informacje, ile dni upłynęło od daty złożonego przez niego zamówienia.
17. Wyświetlić nazwisko klienta, nazwę wydawnictwa i nazwę dostawcy, o ile miejsce zamieszkania klienta, siedziba wydawnictwa oraz siedziba dostawcy są w tym samym mieście.
18. Wyświetlić `id_klienta`, imię i nazwisko tych klientów z województwa podlaskiego, którzy zamówili więcej niż jeden egzemplarz książki.
19. Wyświetlić ile książek zostało zamówionych przez klienta o imieniu 'Anna' lub 'Tadeusz'.
20. Wypisać ISBN, tytuł, imię i nazwisko autora tych książek, których długość tytułu nie przekracza sumy długości imienia i nazwiska tego autora.

Rozdział 5

Funkcje agregujące

W tym rozdziale przedstawiona zostanie logika funkcji agregujących, możliwości modyfikowania ich działania przy użyciu słów kluczowych `GROUP BY` i `HAVING`. Szczegółowe informacje o funkcjach agregujących znajdują się w dokumentacji PostgreSQL pod adresem `https://www.postgresql.org/docs/15/functions-aggregate.html`.

5.1. Funkcje operujące na grupach wierszy

Często celem zapytania jest zrozumienie cech całej kolumny lub tabeli, a nie samo przejrzenie poszczególnych wierszy danych. Język SQL udostępnia funkcje, które można wykorzystać do wykonywania obliczeń na dużych grupach wierszy. Są to następujące funkcje agregujące:

- `COUNT([DISTINCT|ALL] *|kol)` - zwraca liczbę wierszy z kolumny `kol` mających wartość niepustą (różną od `NULL`). Znak `'*'` - powoduje, że wartościowane są wszystkie wiersze grupy nawet te zawierające wartość `NULL` (liczebność grupy), w szczególnym przypadku liczebność tabeli.
- `MAX([DISTINCT|ALL] kol)` - zwraca maksymalną wartość z kolumny `kol` w ramach wierszy grupy. Dla kolumn tekstowych zwracana jest wartość ostatnia w porządku alfabetycznym.
- `MIN([DISTINCT|ALL] kol)` - zwraca minimalną wartość z kolumny `kol` w ramach wierszy grupy. Dla kolumn tekstowych zwracana jest wartość pierwsza w porządku alfabetycznym.
- `SUM([DISTINCT|ALL] kol)` - oblicza i zwraca sumę wartości kolumny `kol` wszystkich wierszy grupy.
- `AVG([DISTINCT|ALL] kol)` - oblicza i zwraca średnią arytmetyczną wartości kolumny `kol` wszystkich wierszy grupy.

- `STDDEV([DISTINCT|ALL] kol)` - zwraca odchylenie standardowe dla próbki na podstawie wszystkich wartości z kolumny `kol`.
- `VARIANCE([DISTINCT|ALL] kol)` - zwraca wariancję dla próbki na podstawie wszystkich wartości z kolumny `kol`.
- `REGR_SLOPE(kol1, kol2)` - zwraca nachylenie linii regresji dla kolumny `kol1` jako zmiennej zależnej i kolumny `kol2` jako zmiennej niezależnej.
- `REGR_INTERCEPT(kol1, kol2)` - zwraca punkt przecięcia z osią linii regresji liniowej dla kolumny `kol1` jako zmiennej zależnej i kolumny `kol2` jako zmiennej niezależnej.
- `CORR(kol1, kol2)` - oblicza i zwraca `współczynnik korelacji Pearsona` dla danych z kolumn `kol1` i `kol2`.

Funkcje agregujące mogą pomóc w sprawnym wykonywaniu różnych zadań:

- Funkcji agregujących można używać dla wszystkich rekordów w tabeli, np.
  - do sprawdzenia liczby rekordów przechowywanych w tabeli:

    ```
    SELECT COUNT(*) AS "Liczba wierszy tabeli"
    FROM t_ksiazka;
    ```

  - do obliczenia wartości wszystkich książek przechowywanych w księgarni:

    ```
    SELECT SUM(cena*ilosc) AS "Razem"
    FROM t_ksiazka;
    ```

  - do obliczenia średniej wartości cen książek:

    ```
    SELECT ROUND(AVG(cena),2) AS "Średnia cena"
    FROM t_ksiazka;
    ```

  - do znalezienia tytułu książki będącej na ostatniej pozycji w porządku alfabetycznym:

    ```
    SELECT MAX(tytul) AS "Ostatni"
    FROM t_ksiazka;
    ```

- Funkcji agregujących można używać również z klauzulą `WHERE`, ograniczając w ten sposób obliczenia do grupy rekordów spełniających warunek tej klauzuli, np.
  - do sprawdzenia ile książek ma cenę wyższą od 25:

    ```
    SELECT COUNT(*) AS "Liczba książek"
    FROM t_ksiazka
    WHERE cena > 25.00;
    ```

  - do sprawdzenia ilu różnych klientów złożyło zamówienia po 1 stycznia 2000 roku:

    ```
    SELECT COUNT(DISTINCT id_klienta) AS "Liczba klientów"
    FROM t_zamowienie
    WHERE data_zam > '2000-01-01';
    ```

– do obliczenia sumy cen książek dostarczanych przez dostawców o identyfikatorach 2 lub 5:

```
SELECT SUM(cena) AS "Suma"
FROM t_ksiazka
WHERE dostawca in (2,5);
```

- Można także łączyć funkcje agregujące za pomocą operacji matematycznych, np. do obliczenia średniej ceny książek można wykorzystać funkcje: `SUM` oraz `COUNT`:

```
SELECT ROUND(SUM(cena)/COUNT(*),2) AS "Średnia"
FROM t_ksiazka;
```

UWAGA:W zależności od argumentów operator dzielenia (/) może zwrócić część całkowitą z dzielenia, gdy argumenty są całkowite lub liczbę rzeczywistą, gdy choć jeden argument jest typu zmiennoprzecinkowego. Zatem, aby wynik dzielenia był liczbą rzeczywistą, wystarczy jeden z jej argumentów pomnożyć przez `1.0` lub zrzutować na typ zmiennoprzecinkowy:
`COUNT(DISTINCT wydawca)*1.0 / COUNT(*)` lub
`COUNT(DISTINCT wydawca)::numeric / COUNT(*)`

5.1.1. Klauzula GROUP BY

Klauzula `GROUP BY` umożliwia podział wierszy relacji na grupy. Wiersze tej samej grupy mają identyczną wartość atrybutu grupowania, który wskazano w klauzuli. Po podziale do każdej z grup można zastosować jedną z funkcji grupowych. Klauzula `GROUP BY` może być stosowana rekurencyjnie w celu wydzielenia podgrup w ramach wcześniej wydzielonych grup. Kolejność dzielenia relacji na grupy i podgrupy odpowiada kolejności kolumn grupowania. Stosowanie funkcji grupowych w klauzuli `SELECT` wyklucza możliwość wyświetlania wartości `n` kolumn pojedynczych wierszy, chyba że nazwy `n` kolumn są wymienione w klauzuli `GROUP BY`.

Na przykład klauzula `GROUP BY` powoduje wydzielenie grupy wierszy z tabeli `t_ksiazka` o tej samej wartości kolumny `wydawca`, a zatem pogrupowanie według identyfikatorów wydawców. Następnie, niezależnie dla każdej grupy, wykonywana jest funkcja grupowania `COUNT(*)`, powodująca zliczenie wierszy w ramach grupy.

```
SELECT wydawca, COUNT(*) AS "Liczba książek"
FROM t_ksiazka
GROUP BY wydawca;
```

W celu wykonania operacji grupowania można też użyć numeru kolumny, zamiast jej nazwy, jak we wcześniejszym przykładzie.

```
SELECT wydawca, COUNT(*) AS "Liczba książek"
FROM t_ksiazka
GROUP BY 1;
```

Wartością klucza w operacji `GROUP BY` może być również wynik wywołania funkcji dla kolumny (lub kolumn), np. można wyodrębnić rok z daty zamówienia i zliczyć liczbę zamówień w danym roku:

```
SELECT TO_CHAR(data_zam,'YYYY'), COUNT(*) AS "Liczba zamówień"
FROM t_zamowienie
GROUP BY TO_CHAR(data_zam,'YYYY');
```

Kolejny przykład ilustruje grupowanie dla kilku kolumn. Najpierw odrzucane są wiersze reprezentujące rok `'1997'`. Potem wiersze grupowane są według identyfikatora wydawcy (kolumna `wydawca`). Następnie, każda z grup jest dzielona na podgrupy według dostawcy (kolumna `dostawca`). Na zakończenie dla każdej podgrupy jest wykonywana funkcja `COUNT()`. Inaczej mówiąc, określamy liczebność różnych grup ze względu na dostawcę, w ramach wcześniej wydzielonych grup według wydawców, po odrzuceniu danych z roku `'1997'`.

```
SELECT wydawca, dostawca, COUNT(*)
FROM t_ksiazka
WHERE rok != '1997'
GROUP BY wydawca, dostawca
ORDER BY wydawca, dostawca DESC;
```

PostgreSQL pozwala również utworzyć kilka kategorii, na podstawie których grupowane są wartości, poprzez zastosowanie w klauzuli `GROUP BY` podklauzuli `GROUPING SETS` postaci:

```
SELECT c1, c2, aggregate_function(c3)
FROM table_name
GROUP BY GROUPING SETS (
  (c1, c2), (c1), (c2), ()
);
```

gdzie `(c1, c2), (c1), (c2), ()` są w tym przypadku czterema możliwymi grupami (kolejność grup nie jest istotna).

Przykładem zastosowania podklauzuli `GROUPING SETS` jest zliczenie książek poszczególnych wydawców, a jednocześnie w tej samej funkcji agregującej określenie łącznej liczby książek w poszczególnych latach u każdego wydawcy. Wiersze zawierające w kolumnie rok wartość `NULL` zawierają informację o łącznej liczbie książek poszczególnych wydawców (rysunek 5.1).

```
SELECT wydawca, rok, COUNT(*)
FROM t_ksiazka
GROUP BY GROUPING SETS (
  (wydawca),
  (wydawca, rok)
)
ORDER BY 1, 2;
```

wydawca	rok	count
1	1978	6
1	1995	3
1	1997	2
1	2007	1
1		12
2	1998	6
2		6
3	2001	6
3		6
4	1997	6
4		6
5	1995	3
5	2001	3
5		6
6	1978	3
6	1995	3
6	1998	3
6		9

(18 rows)

Rysunek 5.1: Wynik zapytania z podklauzulą `GROUPING SET`

Powyższe rozwiązanie można uzyskać również, stosując względem zbiorów wynikowych dwóch zapytań operator zbiorowy `UNION ALL` w następujący sposób:

```
(
 SELECT wydawca, NULL as rok, COUNT(*)
 FROM t_ksiazka
 GROUP BY 1, 2
)
UNION ALL
(
 SELECT wydawca, rok, COUNT(*)
 FROM t_ksiazka
 GROUP BY 1, 2
)
ORDER BY 1, 2;
```

Zostanie zwrócony posortowany zbiór będący sumą zbiorów danych zwracanych przez dwa zapytania: pierwsze zapytanie zwraca zbiór danych pogrupowanych według wydawców, a drugie według wydawców i lat wydania.

Wykorzystanie operatora zbiorowego `UNION ALL` daje takie same wyniki, jak zastosowanie podklazuli `GROUPING SETS`, przy czym to pierwsze rozwiązanie może wymagać pisania bardzo długich zapytań i przez to łatwiej można popełnić błędy.

Opisane do tej pory funkcje agregujące nie wymagały na wejściu posortowanych danych. Istnieją jednak zagregowane statystyki, których obliczanie jest zależne od kolejności wartości. SQL udostępnia zestaw funkcji agregujących dla zbiorów uporządkowanych, takich jak między innymi:

- `MODE()` - zwraca wartość, która występuje najczęściej. Jeśli takich wartości jest kilka, zwracana jest ta, która występuje najwcześniej.
- `PERCENTILE_CONT(pozycja_jako_ułamek)` - zwraca wartość z uporządkowanego zbioru danych o pozycji podanej jako ułamek. W razie potrzeby wartość jest interpolowana na podstawie sąsiednich danych wejściowych.
- `PERCENTILE_DISC(pozycja_jako_ułamek)` - zwraca pierwszą wartość danych wejściowych, której pozycja w uporządkowanym zbiorze danych jest równa lub większa względem podanego ułamka.

Ogólna postać zapytań z użyciem wyżej wymienionych funkcji jest następująca:

```
SELECT funkcja_dla_danych_uporządkowanych
WITHIN GROUP (ORDER BY kolumna_porządkująca)
FROM tabela;
```

Przykładowo, dzięki funkcji `PERCENTILE_CONT` możemy obliczyć medianę cen książek dla wartości funkcji wynoszącej 0.5 (50% w zapisie ułamkowym):

```
SELECT PERCENTILE_CONT(0.5)
WITHIN GROUP (ORDER BY cena) AS Mediana
FROM t_ksiazka;
```

5.1.2. Klauzula HAVING

Czasami interesują nas tylko niektóre wiersze z wyników funkcji agregującej mające określone cechy, dlatego w danych wyjściowych zapytania chcemy zachować tylko te wiersze i usunąć pozostałe. Do filtrowania wyników funkcji agregujących potrzebna jest specjalna klauzula `HAVING` zawsze występująca po klauzuli `GROUP BY`. Pozwala ona na zastosowanie filtru do zagregowanych już danych, a nie do pierwotnego zbioru danych, względem którego stosujemy warunek klauzuli `WHERE`.

Wykorzystamy klauzulę `HAVING` do obliczenia średniej ceny książek dla każdego wydawcy, o ile w księgarni są więcej niż dwie książki przez niego wydane:

```
SELECT wydawca, ROUND(AVG(cena),2)
FROM t_ksiazka
GROUP BY wydawca
HAVING COUNT(1) > 2;
```

5.2. Oczyszczanie danych i sprawdzanie ich jakości

Za pomocą funkcji agregujących można ustalić nie tylko, w których kolumnach brakuje wartości, ale również to, w jaki sposób brakujące dane wpływają na obliczenia i czy przez to kolumny, w których te braki występują, w ogóle nadają się do użytku.

Jednym ze sposobów sprawdzenia, czy w kolumnie znajdują się wartości puste, jest użycie instrukcji `CASE` z funkcjami `SUM` i `COUNT`. W ten sposób dodatkowo wyznaczymy procent brakujących danych.

```
SELECT SUM( CASE WHEN nazwa is NULL OR nazwa in ('Znak', 'PIW')
            THEN 1 ELSE 0 END) :: float / COUNT(*) as "Czy puste?"
FROM t_wydawca;
```

Otrzymany w wyniku zapytania zbiór danych można wykorzystać na różne sposoby, np. można uzupełnić dane, jeśli brakuje ich mniej niż wyznaczona granica lub gdy jest ich więcej, należy wyeliminować kolumnę z analiz, aby otrzymać wiarygodne wnioski.

Jeśli z kolei interesują nas tylko wartości `NULL`, można użyć funkcji `COUNT`, aby wyznaczyć zarówno procent wartości równych `NULL`, jak i tych od niej różnych.

```
SELECT ROUND(COUNT(nazwa)*1.0/COUNT(*),4) as non_null_name,
       1 - ROUND(COUNT(nazwa)*1.0/COUNT(*),4) as null_name
FROM t_wydawca;
```

Innym często wykonywanym zadaniem jest sprawdzanie, czy każda wartość w kolumnie jest unikatowa. Dla kolumn, które są kluczami głównymi, unikatowość jest zapewniona. Nie zawsze jednak możemy dodać dla kolumn ograniczenia `PRIMARY KEY` lub `UNIQUE`. Można wówczas do sprawdzania unikatowości danych wykorzystać funkcje agregujące. Jednym z rozwiązań jest zastosowanie funkcji `COUNT` względem wierszy pogrupowanych według kolumny w klauzuli `GROUP BY` i dodać do tabeli wynikowej tylko te wiersze, których w danej grupie jest więcej niż jeden (klauzula `HAVING`). Jeśli zbiór wynikowy jest pusty, to znaczy, że wszystkie wartości w kolumnie są unikatowe.

```
SELECT id_wydawcy, COUNT(*)
FROM t_wydawca
GROUP BY id_wydawcy
HAVING COUNT(1) > 1;
```

Innym sposobem rozwiązania powyższego zadania może być zastosowanie w klauzuli `SELECT` wyrażenia boolowskiego. Jeżeli zapytanie zwróci wartość `True` to znaczy, że w kolumnie są tylko wartości unikatowe. W przeciwnym razie istnieje co najmniej jedna wartość, która się powtarza.

```
SELECT COUNT(DISTINCT id_wydawcy) = COUNT(*) as equal_ids
FROM t_wydawca;
```

5.3. Funkcja okna

Analizując dane przechowywane w bazie danych, czasami potrzebujemy poznać cechy punktu danych takie jak jego miejsce w zbiorze danych. Typowym przykładem jest pozycja, którą oblicza się zarówno na podstawie samego pomiaru, jak i zbioru danych, w którym się znajduje. W jednym zbiorze danych mogą też występować podgrupy (partycje), na których oparta jest pozycja. W podgrupie należy wybrać wiersze uwzględnione w obliczeniach, aby można było ustalić pozycję na podstawie liczby wierszy przed bieżącym wierszem. Te wybrane wiersze tworzą `okno`. Dla danego zbioru celem jest uzyskanie wyniku dla każdego wiersza. Wynik ten jest zależny zarówno od wartości wiersza, od okna, jak i samego zbioru danych. Funkcja, która służy do wykonywania obliczeń tego rodzaju, to `funkcja okna`.

Podstawowa składnia funkcji okna jest następująca:

```
SELECT [kolumna [, kolumna1],]
funkcja_okna OVER (
  PARTITION BY klucz_podziału
  ORDER BY klucz_porządkowania
)
FROM tabela;
```

Polecenie służy do wyselekcjonowania kolumn wskazanych w klauzuli `SELECT`, które pochodzą z tabeli lub złączeń tabel wskazanych w klauzuli `FROM` z zastosowaniem funkcji okna, którego definicja rozpoczyna się od słowa kluczowego `OVER`. Definicja funkcji okna zawiera kolumnę lub kolumny wykorzystywane do podziału (`PARTITION BY`) oraz kolumnę lub kolumny służące do porządkowania danych (`ORDER BY`).

Jako funkcje okna można stosować dowolne funkcje agregujące. Rozważmy przykład zastosowania funkcji `COUNT` w roli funkcji okna, która służy do wyświetlenia identyfikatora, imienia i nazwiska klientów wraz z liczbą wszystkich klientów w osobnej kolumnie dla każdego z nich.

```
SELECT id_klienta, imie, nazwisko,
COUNT(*) OVER () as Razem
FROM t_klient
ORDER by id_klienta;
```

```
 id_klienta |   imie    |  nazwisko  | razem
------------+-----------+------------+-------
          1 | Jan       | Kowalski   |     5
          2 | Tadeusz   | Malinowski |     5
          3 | Krystyna  | Torbicka   |     5
          4 | Anna      | Marzec     |     5
          5 | Adam      | Koper      |     5
(5 rows)
```

Rysunek 5.2: Wynik zapytania z `COUNT` jako funkcją okna

Na rysunku 5.2 przedstawiamy wynik działania powyższego zapytania, w którym dodana została kolumna z informacją o liczbie wszystkich klientów. Wartość ta w każdym wierszu odpowiada wynikowi zapytania:

```
SELECT COUNT(*)
FROM t_klient;
```

Teraz wykorzystamy funkcję okna i jej klauzulę `PARTITION BY` względem kolumny `miasto`, aby SQL dokonał podziału zbioru danych na wiele podgrup na podstawie unikatowych wartości tej kolumny. Dla każdej podgrupy obliczamy następnie liczbę wierszy za pomocą funkcji `COUNT(*)`. Dzięki temu dla każdego miasta w kolumnie `Razem` wyświetlona zostanie liczba wierszy z tą wartością, patrz rysunek 5.3.

```
SELECT id_klienta, imie, nazwisko, miasto,
COUNT(*) OVER (PARTITION BY miasto) as Razem
FROM t_klient
ORDER by id_klienta;
```

```
 id_klienta |   imie    |  nazwisko  |  miasto  | razem
------------+-----------+------------+----------+-------
          1 | Jan       | Kowalski   | Warszawa |     3
          2 | Tadeusz   | Malinowski | Warszawa |     3
          3 | Krystyna  | Torbicka   | Warszawa |     3
          4 | Anna      | Marzec     | Lipsk    |     2
          5 | Adam      | Koper      | Lipsk    |     2
(5 rows)
```

Rysunek 5.3: Wynik zapytania z klauzulą `PARTITION BY`

Zastosujemy teraz klauzulę `ORDER BY` w funkcji okna porządkującą po `id_klienta`.

```
SELECT id_klienta, imie, nazwisko, miasto,
COUNT(*) OVER (ORDER BY id_klienta) as Razem
FROM t_klient
ORDER by id_klienta;
```

```
 id_klienta |   imie   |  nazwisko  |  miasto  | razem
------------+----------+------------+----------+-------
          1 | Jan      | Kowalski   | Warszawa |     1
          2 | Tadeusz  | Malinowski | Warszawa |     2
          3 | Krystyna | Torbicka   | Warszawa |     3
          4 | Anna     | Marzec     | Lipsk    |     4
          5 | Adam     | Koper      | Lipsk    |     5
(5 rows)
```

Rysunek 5.4: Wynik zapytania z klauzulą `ORDER BY`

Na rysunku 5.4 przedstawiamy wyniki powyższego zapytania, w których w kolumnie `Razem` widać narastającą sumę, która określa łączną liczbę klientów. Ponieważ nie ma zdefiniowanej klauzuli `PARTITION BY`, w obliczeniach uwzględniany jest cały zbiór danych. Gdy w zbiorze danych nie ma też określonej klauzuli `ORDER BY`, używane jest tylko jedno okno obejmujące cały zbiór. Natomiast jeśli użyjemy klauzuli `ORDER BY`, wiersze w grupie będą porządkowane zgodnie z nią i dla każdej unikatowej wartości z tak uporządkowanych danych SQL utworzy grupę dotyczącą danej wartości. Grupa zawiera wszystkie wiersze, w których ta wartość występuje. Zapytanie tworzy okno dla każdej takiej grupy obejmujące wszystkie wiersze z danej grupy i wszystkie wiersze poprzednie. W przykładzie z rysunku 5.4 zbiór danych porządkowany jest według klucza głównego tabeli `t_klient`. Dlatego każdy wiersz, mając unikatową wartość, tworzy odrębną grupę. Przed pierwszą grupą, składającą się tylko z jednego, pierwszego wiersza, nie występują żadne inne wiersze, dlatego tworzone jest odrębne, jednowierszowe okno. Okno dla drugiej grupy obejmuje tę grupę i poprzedni wiersz. W ten sposób narastająco dla każdej grupy tworzone są kolejne okna. Po ustaleniu okien dla każdej grupy obliczana jest liczba wierszy wchodzących w jej skład. Wynik przypisywany jest do wierszy z poszczególnych grup wartości.

Teraz w funkcji okna użyjemy obydwu klauzul: `PARTITION BY` i `ORDER BY`.

```
SELECT id_klienta, imie, nazwisko, miasto,
COUNT(*) OVER (
  PARTITION BY miasto
  ORDER BY id_klienta
) as Razem
FROM t_klient
ORDER by id_klienta;
```

```
 id_klienta |   imie   |  nazwisko  |  miasto  | razem
------------+----------+------------+----------+-------
          1 | Jan      | Kowalski   | Warszawa |     1
          2 | Tadeusz  | Malinowski | Warszawa |     2
          3 | Krystyna | Torbicka   | Warszawa |     3
          4 | Anna     | Marzec     | Lipsk    |     1
          5 | Adam     | Koper      | Lipsk    |     2
(5 rows)
```

Rysunek 5.5: Wynik zapytania z klauzulą `PARTITION BY` i `ORDER BY`

Na rysunku 5.5 pokazujemy wyniki działania przedstawionego wyżej zapytania, gdzie w obrębie danej grupy podzielonej według miasta otrzymaliśmy coś w rodzaju rankingu. Klauzula `PARTITION BY` dzieli zbiór danych na grupy (nazywane tu podgrupami lub partycjami) na podstawie wartości z kolumny `miasto`, a następnie każda podgrupa jest używana do zliczania wierszy i każda ma własny zestaw grup wartości. Te grupy wartości są uporządkowane w ramach podgrupy i na ich podstawie tworzone są okna, wyznaczana jest ich kolejność i uruchamiana funkcja okna. Ostateczne wyniki są przypisywane do poszczególnych wierszy grup wartości.

Zastosujemy teraz składnię polecenia `SELECT` z klauzulą `WINDOW`, która skraca i upraszcza tworzenie aliasów okien. Na początku rozważmy zapytanie, które wykorzysta to samo okno dla różnych funkcji do obliczenia narastającej liczby klientów oraz sumy narastającej liczby klientów, którzy w nazwisku na dowolnym miejscu mają literę `'k'` (bez rozróżniania wielkości liter).

```
SELECT id_klienta, imie, nazwisko, miasto,
COUNT(*) OVER (
  PARTITION BY miasto
  ORDER BY id_klienta
) as Razem_Count,
SUM(CASE WHEN nazwisko ILIKE '%k%' THEN 1 ELSE 0 END) OVER (
  PARTITION BY miasto
  ORDER BY id_klienta
) as Razem_Sum
FROM t_klient
ORDER by id_klienta;
```

Upraszczamy powyższy kod, stosując klauzulę `WINDOW`.

```
SELECT id_klienta, imie, nazwisko, miasto,
COUNT(*) OVER w as Razem_Count,
SUM(CASE WHEN nazwisko ILIKE '%k%' THEN 1 ELSE 0 END)
OVER w as Razem_Sum
FROM t_klient
```

```
WINDOW w AS (
  PARTITION BY miasto
  ORDER BY id_klienta
)
ORDER by id_klienta;
```

```
 id_klienta |   imie   |  nazwisko  |  miasto  | razem_count | razem_sum
------------+----------+------------+----------+-------------+-----------
          1 | Jan      | Kowalski   | Warszawa |           1 |         1
          2 | Tadeusz  | Malinowski | Warszawa |           2 |         2
          3 | Krystyna | Torbicka   | Warszawa |           3 |         3
          4 | Anna     | Marzec     | Lipsk    |           1 |         0
          5 | Adam     | Koper      | Lipsk    |           2 |         1
(5 rows)
```

Rysunek 5.6: Wynik zapytania z klauzulą `WINDOW`

Na rysunku 5.6 przedstawiamy wyniki działania zapytania z użyciem klauzuli `WINDOW` (takie same jak, w przypadku zapytania bez użycia tej klauzuli). W przypadku, gdy `nazwisko` nie spełnia warunku wyboru w funkcji `SUM`, w kolumnie `razem_sum` pojawia się zero.

Oprócz funkcji agregujących, które mogą wystąpić jako funkcje okna, w tej roli mogą również wystąpić różne funkcje statystyczne wymienione w tabeli 5.1. Zwykle po wywołaniu dowolnej z tych funkcji występuje słowo kluczowe `OVER`, a po nim w nawiasach okrągłych opcjonalne klauzule `PARTITION BY` oraz `ORDER BY`.

Tabela 5.1: Statystyczne funkcje okna

Funkcja	Objaśnienie
`ROW_NUMBER`	Określa numer bieżącego wiersza w grupie; numerowanie rozpoczyna się od 1.
`RANK`	Zwraca pozycję wiersza w grupie; nie tworzy luk.
`DENSE_RANK`	Zwraca pozycję wiersza w grupie; tworzy luki.
`LAG`	Zwraca wartość uzyskaną dla wiersza grupy poprzedzającego bieżący o przesunięcie.
`LEAD`	Zwraca wartość uzyskaną dla wiersza grupy następującego po bieżącym o przesunięcie.
`NTILE`	Dzieli wiersze w grupie na możliwie równe przedziały i przypisuje do każdego wiersza liczby całkowite od 1 do wartości argumentu.

Jako przykład zdefiniujemy zapytanie, które używa funkcji `RANK` do uporządkowania klientów zgodnie z datą zamówień z podziałem według miasta zamieszkania klienta. Na rysunku 5.7 przedstawiamy wyniki tego zapytania.

```
SELECT id_klienta, imie, nazwisko, miasto, data_zam,
RANK() OVER w as klient_rank
FROM t_klient JOIN t_zamowienie USING(id_klienta)
WINDOW w AS (
  PARTITION BY miasto
  ORDER BY data_zam
)
ORDER by id_klienta;
```

id_klienta	imie	nazwisko	miasto	data_zam	klient_rank
1	Jan	Kowalski	Warszawa	2007-01-10	4
1	Jan	Kowalski	Warszawa	2012-05-12	6
2	Tadeusz	Malinowski	Warszawa	2002-01-11	1
2	Tadeusz	Malinowski	Warszawa	2004-01-10	3
3	Krystyna	Torbicka	Warszawa	2003-01-10	2
3	Krystyna	Torbicka	Warszawa	2011-01-12	5
4	Anna	Marzec	Lipsk	2001-01-11	1
4	Anna	Marzec	Lipsk	2007-10-12	2
4	Anna	Marzec	Lipsk	2010-01-12	4
4	Anna	Marzec	Lipsk	2012-04-12	6
5	Adam	Koper	Lipsk	2008-01-11	3
5	Adam	Koper	Lipsk	2012-01-12	5

(12 rows)

Rysunek 5.7: Uporządkowane dane według daty złożenia zamówienia

5.4. Zadania do samodzielnego rozwiązania

1. Wyświetlić minimalną, średnią i maksymalną cenę książek.
2. Wyświetlić minimalną i maksymalną cenę książek u poszczególnych wydawców.
3. Wyświetlić liczbę książek dostarczonych przez dostawcę o identyfikatorze 1.
4. Wyświetlić średnią cenę i średnią cenę wszystkich egzemplarzy książek poszczególnych wydawców. Dane uporządkować zgodnie z rosnącą wartością średniej ceny książki.
5. Wyświetlić różnicę między najwyższą i najniższą ceną książki.
6. Wyświetlić nazwy wydawców, którzy wydali więcej niż 3 książki.
7. Sprawdzić, czy numery książek (`ISBN`) powtarzają się.
8. Wyświetlić najniższą cenę książki w grupie każdego autora. Pominąć grupy, w których minimalna cena jest niższa od 25. Wyniki uporządkować według rosnących cen.
9. Wyświetlić liczbę książek wysłanych w poszczególnych miesiącach roku.
10. Obliczyć i wyświetlić maksymalną cenę książek wydanych w dowolnym roku, w którym wydano również książki Cypriana Norwida.

11. Wyświetlić `id_klienta`, imię i nazwisko tych klientów z województwa mazowieckiego, którzy zamówili dokładnie jedną książkę.
12. Wyświetlić każde imię autora, które się powtarza.
13. Wyświetlić identyfikator każdego zamówienia, którego wartość jest wyższa od 200 złotych.
14. Wyświetlić najniższą cenę książki w grupie każdego autora. Pominąć grupy, w których minimalna cena jest niższa od 25. Wyniki uporządkować według rosnących cen.
15. Wyświetlić dane o autorach tych książek, które nigdy nie zostały sprzedane.
16. Wyświetlić, ile książek zostało sprzedanych (wziąć pod uwagę tylko zrealizowane zamówienia) w poszczególnych miesiącach roku (wziąć pod uwagę datę zamówienia).
17. Wyświetlić minimalną cenę książek wydanych w roku innym niż: `'1978', '1995' lub '1998'`.
18. Wyświetlić ISBN, tytuł, imię i nazwisko autora tych książek, które zamówili klienci o imieniu i nazwisku różnej długości. (np. `Adam Kowalski`).
19. Wyświetlić najstarsze alfabetycznie imię autora, które się powtarza.
20. Wyświetlić identyfikator zamówienia z najwcześniejszą datą realizacji zamówienia (kolumna `data_wys`).

Rozdział 6

Zagnieżdżanie zapytań

W relacyjnym modelu danych każde zapytanie zwraca dwuwymiarową tabelę wynikową. Istnieje sposób na wykorzystanie danych z tabel wynikowych w innych zapytaniach. Wystarczy umieścić zapytanie w nawiasie okrągłym w odpowiedniej klauzuli polecenia `SELECT`, nadać jej alias, gdy jest taka potrzeba i tak otrzymane dane poddawać dalszej analizie. Często tego typu zapytania nazywa się podrzędnymi instrukcjami `SELECT` albo podzapytaniami, ponieważ pozwalają one na wykonywanie instrukcji `SELECT` w obrębie dowolnej instrukcji SQL.

Podzapytania możemy stosować praktycznie w dowolnym bloku logicznym zapytania. Jedynym ograniczeniem jest rodzaj zwracanego zbioru, który musi pasować do miejsca, w którym chcemy go użyć. Na przykład w klauzuli `FROM`, może to być dowolny zbiór (jednoelementowy, wieloelementowy itd.), z kolei w klauzuli `SELECT` musi to być wartość skalarna, czyli zbiór jednoelementowy opisany jednym atrybutem.

W tym rozdziale omówione zostaną mechanizmy pozwalające na przeprowadzenie w zapytaniu analiz na podstawie wyników zapytań w nich zagnieżdżonych.

6.1. Kategorie podzapytań

Podzapytania możemy podzielić na dwie kategorie ze względu na powiązanie z zapytaniem nadrzędnym:

- **niezależne** – podzapytanie jest wykonywane jako pierwsze, jednokrotnie, a jego wyniki są przekazywane do zapytania zewnętrznego;
- **skorelowane** – najpierw wykonywane jest zapytanie nadrzędne i dla każdego wiersza tego zapytania wykonywane jest podzapytanie z nim skorelowane. W zapytaniu skorelowanym konieczne jest zastosowanie aliasów relacji, na których operuje zapytanie nadrzędne i odwoływanie się do nich w podzapytaniu.

6.1.1. Podzapytania niezależne

Wykorzystamy fakt zwracania przez podzapytanie zbioru danych, który potraktujemy jako w pełni określoną nazwaną tabelę występującą w klauzuli `FROM`. Stąd konieczność stosowania aliasów oraz unikalnych nazw kolumn w ramach podzapytań.

Jako przykład wyświetlimy, używając podzapytania niezależnego, te imiona i nazwiska klientów, którzy mieszkają w Warszawie z ograniczeniem do nazwisk zaczynających się na jedną z liter od 'A' do 'K'.

```
SELECT *
FROM (
  -- wstępna selekcja danych
  -- może tu być dowolne, nawet i skomplikowane, zapytanie
  SELECT imie, nazwisko
  FROM t_klient
  WHERE miasto = 'Warszawa'
) AS my_question_1
WHERE nazwisko ~ '[A-K].*';
```

W kolejnym przykładzie komplikujemy podzapytanie, wykorzystując funkcję agregującą do zliczenia średniej w dwóch wybranych latach, by potem wykorzystać wcześniejsze dane do wyświetlenia tylko tych lat, dla których średnia spełnia podany warunek.

```
SELECT rok
FROM (
  SELECT rok, AVG(cena) AS Average
  FROM t_ksiazka
  WHERE rok in ('1997','2001')
  GROUP BY rok
) AS my_question_2
WHERE Average > 25;
```

Zauważmy, że w każdej chwili możemy to podzapytanie uruchomić niezależnie od zapytania nadrzędnego. Wykonane zostanie raz w trakcie całego procesu logicznego przetwarzania tego zapytania.

Wyznaczamy teraz średnią wartość dla wszystkich zamówień.

```
SELECT ROUND(AVG(t_ksiazka.ilosc*t_ksiazka.cena),2) as AVG
FROM t_ksiazka INNER JOIN t_z_ksiazka USING (ISBN);
```

Zwrócony zbiór danych możemy umieścić w każdym miejscu zapytania – jako podzapytanie. Najczęściej będziemy go stosować w warunkach klauzuli `WHERE` lub w klauzuli

SELECT. Może być też stosowany w innych miejscach, gdzie tworzymy wyrażenia, np. do filtrowania grup w klauzuli HAVING czy w warunkach złączeń w ON.

Wykorzystamy wcześniej uzyskany zbiór danych, aby odfiltrować rekordy w klauzuli WHERE i dodatkowo wyświetlić je w klauzuli SELECT jako wartość dodatkowej kolumny.

```
SELECT id_zam, data_zam,
(
  SELECT ROUND(AVG(t_ksiazka.ilosc*t_ksiazka.cena),2) as AVG
  FROM t_ksiazka INNER JOIN t_z_ksiazka USING (ISBN)
) AS AVG_Total
FROM t_zamowienie
WHERE karta != 1
AND data_zam BETWEEN '1999-06-01' AND '2024-05-30'
AND 25 < (
    SELECT ROUND(AVG(t_ksiazka.ilosc*t_ksiazka.cena),2) as AVG
    FROM t_ksiazka INNER JOIN t_z_ksiazka USING (ISBN)
);
```

Gdy podzapytanie wyznacza jeden wiersz (zwykle ograniczony do pojedynczej kolumny w klauzuli SELECT podzapytania), w warunku selekcji zapytania nadrzędnego stosujemy najczęściej jeden z operatorów porównania, np. $=, >=, <$. Gdy podzapytanie wyznacza więcej niż jeden wiersz, w warunku selekcji zapytania nadrzędnego stosujemy najczęściej operator IN.

Dodatkowo dla podzapytań wprowadzono operatory:

- ALL – powoduje porównanie pojedynczej wartości z każdą wartością wyznaczoną przez podzapytanie. Warunek selekcji zapytania nadrzędnego jest spełniony, jeżeli wszystkie wartości listy spełniają ten warunek.
- ANY – powoduje porównanie pojedynczej wartości (umieszczonej po jego lewej stronie) z każdą wartością wyznaczoną przez podzapytanie. Warunek selekcji zapytania nadrzędnego jest spełniony, jeżeli lista wartości wyznaczonych przez podzapytanie zawiera choć jeden element spełniający ten warunek.
- EXISTS – sprawdza, czy podzapytanie zwraca niepusty wynik i zwraca True jeśli tak jest, w przeciwnym przypadku zwraca False.
- NOT EXISTS – sprawdza, czy podzapytanie jest puste i zwraca True jeśli tak jest, w przeciwnym przypadku zwraca False.

Ważne jest, aby liczba wartości wyznaczonych przez podzapytanie oraz ich typ były zgodne z liczbą i typem kolumn użytych w warunku selekcji zapytania nadrzędnego.

Poniższy przykład wyświetla wszystkie informacje o książkach, których cena jest najniższa.

```
SELECT *
FROM t_ksiazka
WHERE cena = (
    SELECT MIN(cena)
    FROM t_ksiazka
);
```

Kolejny przykład pokazuje te tytuły i identyfikatory wydawców książek, które dostarczył dostawca o nazwie 'UPS'.

```
SELECT tytul, wydawca
FROM t_ksiazka
WHERE dostawca = (
    SELECT id_dostawcy
    FROM t_dostawca
    WHERE nazwa = 'UPS'
);
```

W przeciwieństwie do wcześniejszych przykładów, w których podzapytania zwróciły jeden wiersz, pokażemy teraz zastosowanie operatora IN, gdy zapytanie zwraca więcej rekordów oraz wykorzystanie więcej niż jednej nazwy kolumny (krotki, pary) do porównywania z parą zwracanych wartości w jednym rekordzie podzapytania. Wyświetlamy wszystkie informacje o książkach, których cena jest minimalną ceną w grupie każdego dostawcy.

```
SELECT *
FROM t_ksiazka
WHERE (cena,dostawca) IN (
    SELECT MIN(cena), dostawca
    FROM t_ksiazka
    GROUP BY dostawca
);
```

Powyższe zapytanie można równoważnie zapisać używając operatora ANY.

```
SELECT *
FROM t_ksiazka
WHERE (cena,dostawca) = ANY (
    SELECT MIN(cena), dostawca
    FROM t_ksiazka
    GROUP BY dostawca
);
```

Wyświetlimy teraz wybrane informacje o książkach, których cena jest większa od każdej ceny książki dostarczonej przez 'UPS'.

```
SELECT ISBN, tytul, wydawca, dostawca
FROM t_ksiazka
WHERE cena > ALL (
    SELECT cena
    FROM t_ksiazka inner join t_dostawca on dostawca = id_dostawcy
    WHERE nazwa = 'UPS'
);
```

Podzapytania można zagnieżdżać również w klauzuli `HAVING` w celu odrzucenia (przyjęcia) określonych grup krotek w zależności od wyniku podzapytania.

Jako przykład wyświetlimy informacje o całkowitej wartości zamówień poszczególnych klientów, przy czym odrzucamy te rekordy, dla których całkowita wartość zamówień klienta jest mniejsza lub równa średniej wartości wszystkich zamówień.

```
SELECT id_klienta, SUM(tk.cena*tzk.ilosc) AS "Cena całkowita"
FROM t_zamowienie INNER JOIN t_z_ksiazka tzk USING (id_zam)
     INNER JOIN t_ksiazka tk USING (ISBN)
GROUP BY id_klienta
HAVING SUM(tk.cena*tzk.ilosc) > (
  SELECT ROUND(AVG(t_ksiazka.ilosc*t_ksiazka.cena),2)
  FROM t_ksiazka INNER JOIN t_z_ksiazka USING (ISBN)
);
```

6.1.2. Podzapytania skorelowane

Drugim typem podzapytań są podzapytania skorelowane, czyli bezpośrednio powiązane z zapytaniem nadrzędnym. Łącznikiem jest jedna lub więcej kolumn, przekazywanych z zapytania nadrzędnego. Konstrukcja podzapytania skorelowanego zazwyczaj (ale nie zawsze) pociąga za sobą konieczność wielokrotnego wykonania podzapytania dla każdego wiersza rozpatrywanego w zapytaniu nadrzędnym.

W poniższym podzapytaniu jest wyznaczana przeciętna cena książek tego samego wydawcy co książka analizowana przez zapytanie nadrzędne.

```
SELECT tytul, cena, wydawca
FROM t_ksiazka tk
WHERE cena > ( SELECT AVG(cena)
               FROM t_ksiazka
               WHERE wydawca = tk.wydawca
);
```

Zapytanie wyznacza ISBN książki oraz identyfikator autora, o ile na stanie księgarni mamy tylko i wyłącznie jeden przez niego napisany tytuł. Jest to ilustracja sposobu użycia operatora `NOT EXISTS`.

```
SELECT ISBN, id_autora
FROM t_a_ksiazki tak
WHERE NOT EXISTS ( SELECT ISBN
                   FROM t_a_ksiazki
                   WHERE id_autora = tak.id_autora and ISBN != tak.ISBN
) ORDER BY 2;
```

Aby sprawdzić poprawność uzyskanych wyników z powyższego zapytania, można wyświetlić identyfikatory tych autorów, dla których liczba numerów książek (`ISBN`) wynosi 1.

```
SELECT id_autora, COUNT(ISBN)
FROM t_a_ksiazki
GROUP BY id_autora
HAVING COUNT(ISBN)=1
ORDER BY 1;
```

W wielu przypadkach, samodzielnie bądź systemowo (działanie optymalizatora zapytań), podzapytania skorelowane można sprowadzić do zwykłego łączenia tabel. Przykładowo rozważmy zapytanie, które wyświetla niepowtarzające się tytuły książek wydanych po roku 1995 i dostarczonych przez dostawcę o nazwie 'Stolica', których tytuły zaczynają się od jednej z wymienionych liter.

```
SELECT distinct tytul
FROM t_ksiazka tk
WHERE tk.rok > '1995' AND tk.dostawca IN (
        SELECT td.id_dostawcy
        FROM t_dostawca td
        WHERE td.nazwa = 'Stolica'
        AND lower(substring(tk.tytul,1,1)) in ('a','b','d')
);
```

Zapytanie takie można przepisać, wykorzystując łączenie tabel.

```
SELECT distinct tytul
FROM t_ksiazka, t_dostawca
WHERE t_ksiazka.rok > '1995'
    AND t_ksiazka.dostawca = t_dostawca.id_dostawcy
    AND t_dostawca.nazwa = 'Stolica'
    AND lower(substring(t_ksiazka.tytul,1,1)) in ('a','b','d');
```

Istnieje jeszcze jedna transformacja zapytań skorelowanych wykorzystująca definiowanie dopełnienia dla zbioru danych zwracanych przez podzapytanie skorelowane.

Rozważmy zatem zapytanie skorelowane, które wyświetla tytuł najdroższej książki w miękkiej oprawie i identyfikator jej dostawcy.

```
SELECT distinct x.tytul, x.dostawca
FROM t_ksiazka x
WHERE   x.oprawa = 'miękka'
        AND x.cena >= ALL ( SELECT y.cena
                            FROM t_ksiazka y
                            WHERE x.dostawca = y.dostawca
                            AND y.oprawa ='miękka' );
```

Zgodnie z kolejnością operacji w transformacji obliczamy dopełnienie.

```
SELECT distinct x.tytul, x.dostawca
FROM t_ksiazka x
WHERE   x.oprawa = 'miękka'
        AND x.cena < ANY (  SELECT y.cena
                            FROM t_ksiazka y
                            WHERE x.dostawca = y.dostawca
                            AND y.oprawa = 'miękka' );
```

Następnie przepisujemy zapytanie, wykorzystując łączenie tabel.

```
SELECT distinct x.tytul, x.dostawca
FROM t_ksiazka x, t_ksiazka y
WHERE   x.oprawa = 'miękka' AND x.dostawca = y.dostawca
        AND y.oprawa = 'miękka' AND x.cena < y.cena;
```

I na końcu odejmujemy od zbioru wszystkich książek w miękkiej oprawie te z nich, dla których istnieją książki w miękkiej oprawie o wyższej cenie.

```
  ( SELECT x.tytul, x.dostawca
    FROM t_ksiazka x
    WHERE x.oprawa = 'miękka' )
EXCEPT --(MINUS)
  ( SELECT x.tytul, x.dostawca
    FROM t_ksiazka x, t_ksiazka y
    WHERE   x.oprawa = 'miękka'
            AND x.dostawca = y.dostawca
            AND y.oprawa = 'miękka'
            AND x.cena < y.cena );
```

6.2. Zagnieżdżanie rekurencyjne

Podzapytania niezależne (w przeciwieństwie do zapytań skorelowanych), bez względu na głębokość zagnieżdżenia, są zawsze wykonywane w kolejności od najbardziej zagnieżdżonego do najbardziej zewnętrznego. Każde podzapytanie ograniczamy nawiasami i zagnieżdżamy po prawej stronie warunku zapytania nadrzędnego (zapytania umieszczonego o jeden poziom wyżej w strukturze zagłębienia). Ponadto przy zagnieżdżaniu wielopoziomowym:

- liczba oraz typ kolumn występujących w klauzuli `SELECT` podzapytania musi być zgodna z liczbą i typem kolumn użytych w warunku zapytania nadrzędnego (zapytania wyższego poziomu zagnieżdżenia);
- w podzapytaniach nie używamy klauzuli `ORDER BY`; klauzula `ORDER BY` może wystąpić wyłącznie jako ostatnia klauzula najbardziej zewnętrznego zapytania;
- w podzapytaniu można używać operatorów zbiorowych; w warunkach zapytań zewnętrznych, poza operatorami `ANY`, `ALL` `EXISTS` i `NOT EXISTS`, można stosować dowolne operatory języka SQL.

Jako przykład rozważmy zapytanie wyświetlające wszystkie informacje o książkach, których ceny są większe niż najwyższa cena książki dostarczanej przez `'UPS'`.

```
SELECT *
FROM t_ksiazka
WHERE cena > (
    SELECT MAX(cena)
    FROM t_ksiazka
    WHERE dostawca = (
        SELECT id_dostawcy
        FROM t_dostawca
        WHERE nazwa = 'UPS' ));
```

Jako pierwszy wyznaczany jest identyfikator dostawcy o nazwie 'UPS'. Następnie jest on wykorzystywany do wyznaczenia najwyższej ceny dostarczanych przez niego książek, aby z kolei wyświetlić wszystkie informacje o książkach, których cena jest wyższa od wyznaczonej maksymalnej kwoty.

6.3. Wyrażenia WITH

Wspólne wyrażenia tablicowe (ang. *Common Tabel Expressions* - `CTE`) są innym rodzajem podzapytań i można stosować je praktycznie wszędzie – w widokach, funkcjach, procedurach składowanych, skryptach itp.. Pozwalają tworzyć tabele tymczasowe z użyciem

klauzuli WITH. Upraszczają i poprawiają czytelność kodu SQL, a ich stosowanie nie ma wpływu na wydajność zapytań. Struktura CTE pozwala na realizację rekurencji, przy czym w tym podręczniku nie będziemy omawiali tego zagadnienia. Zainteresowanego czytelnika odsyłamy do dokumentacji PostgreSQL pod linkiem https://www.postgresql.org/docs/15/queries-with.html.

Definicję CTE otwiera słowo kluczowe WITH z nazwą zbioru, który zostanie za jego pomocą utworzony. Do tej nazwanej, tymczasowej tabeli będziemy odwoływać się w zapytaniu, tak jak do zwykłej tabeli. W przykładzie zdefiniujemy wspólne wyrażenie tablicowe o nazwie my_stat, które wyznacza 5 pierwszych rekordów spośród wszystkich zawierających informacje o identyfikatorze klienta, liczbie jego wszystkich zamówień oraz dacie ostatniego zamówienia. W wyrażeniu tym każdej kolumnie nadawana jest unikatowa nazwa. Nazwy kolumnom można również nadać w „ciele" wyrażenia CTE jako aliasy i nie jest wówczas wymagane ich jawne nazywanie w klauzuli WITH. Do raz zdefiniowanego CTE, możemy się odnosić wiele razy w zapytaniu, z którym jest związany. Zakres widoczności jest bardzo wąski i obejmuje tylko i wyłącznie zapytanie następujące po nim, ale w pełnym zakresie. Dokładnie tak jakbyśmy odpytywali zbiór, tabelę lub widok, określoną w tym przypadku przez CTE. W naszym przykładzie zbiór danych wyznaczonych przez wyrażenie CTE zostanie wykorzystany do wyświetlenia zamiast identyfikatora klienta, który służy do łączenia tabel, jego imienia i nazwiska wraz z informacją o liczbie wszystkich zamówień tego klienta i dacie jego ostatniego zamówienia.

```
-- definicja wyrażenia tablicowego o nazwie my_stat
WITH my_stat (id_klienta, Total_number, Max_date)
AS (
  -- można uruchomić testowo tylko zawartość CTE,
  -- aby sprawdzić, co zwraca i jakich danych będziemy potem używać
  SELECT id_klienta, COUNT(*), MAX(data_zam)
  FROM t_zamowienie
  GROUP BY id_klienta
  ORDER BY 2
  LIMIT 5
)
-- bezpośrednio po definicji zapytanie odwołujące się m.in. do my_stat
SELECT tk.imie, tk.nazwisko, ms.Total_number, ms.Max_date
FROM my_stat ms left join t_klient tk ON ms.id_klienta = tk.id_klienta;
```

Za pomocą klauzuli WITH prezentowane na stronie 79 zapytanie (wyświetlające te imiona i nazwiska klientów, którzy mieszkają w Warszawie z ograniczeniem do nazwisk zaczynających się na jedną z liter od 'A' do 'K') można zapisać w następujący, bardziej czytelny sposób:

```
-- definicja wyrażenia tablicowego o nazwie my_question
WITH my_question AS (
  SELECT imie, nazwisko
  FROM t_klient
  WHERE miasto = 'Warszawa'
)
SELECT *
FROM my_question
WHERE nazwisko ~ '[A-K].*';
```

Możemy definiować kilka wyrażeń tablicowych i co istotne, widoczne są one nie tylko w zapytaniu powiązanym z `WITH`, ale również w kolejno określanych `CTE` (jeśli jest ich więcej niż 1).

```
WITH
CTE_1 AS (
  SELECT id_klienta, COUNT(*) num_Order, MAX(data_zam) Max_date
  FROM t_zamowienie
  GROUP BY id_klienta ),
CTE_2 AS (
  SELECT id_klienta, num_Order, Max_date
  FROM CTE_1
  WHERE num_Order < 4 )
SELECT CTE_1.*, CTE_2.*
FROM CTE_1 LEFT JOIN CTE_2 USING (id_klienta);
```

I ostatni przykład użycia klauzuli `WITH`, dodatkowo z wykorzystaniem funkcji okna `RANK()` z klauzulą `PARTITION BY` przypisującą każdemu rekordowi w obrębie grupy numer pozycji, którą w tej grupie zajmuje zgodnie z pewnym wyznaczonym porządkiem. (Patrz dokumentacja PostgreSQL: `https://www.postgresql.org/docs/15/tutorial-window.html`.)

```
WITH CTE_3 AS (
  SELECT id_klienta, id_zam, data_zam, RANK() OVER (
      PARTITION BY id_klienta
      ORDER BY data_zam DESC
     ) data_zam_rank
  FROM t_zamowienie
)
SELECT imie, nazwisko, CTE_3.*
FROM t_klient INNER JOIN CTE_3 USING (id_klienta);
```

6.4. Podzapytania i funkcje agregujące w praktyce

Przedstawimy teraz kilka przykładów zastosowania funkcji agregujących oraz podzapytań w instrukcjach wykorzystujących istniejące dane.

Utworzymy tabelę z istniejących danych o nazwie `New_table` zawierającą dane z kolumn `id_klienta`, `imie` i `nazwisko` tych klientów, których nazwisko zawiera literę 'o' na dowolnym miejscu bez rozróżniania wielkości liter.

```
CREATE TABLE New_table AS
SELECT id_klienta, imie, nazwisko
FROM t_klient
WHERE nazwisko ILIKE '%o%';
```

Dodajemy do tabeli `New_table` kolumnę `kwota` typu liczba rzeczywista 6 cyfrowa z 2 miejscami po przecinku.

```
ALTER TABLE New_table ADD kwota numeric(6,2);
```

Do tabeli `New_table` dodajemy, o ile nie istnieją, tych klientów, którzy złożyli zamówienie po '2008-01-01' i nie mają na imię 'Adam'.

```
INSERT INTO New_table
SELECT DISTINCT id_klienta, imie, nazwisko
FROM t_klient JOIN t_zamowienie USING(id_klienta)
WHERE id_klienta NOT IN (SELECT id_klienta FROM New_table)
      AND imie != 'Adam'
      AND data_zam > '2008-01-01';
```

Dla każdego klienta o identyfikatorze `id_klienta` z tabeli `New_table` uaktualniamy dane w kolumnie `kwota` do całkowitego kosztu wszystkich jego zamówień, które zostały zrealizowane.

```
UPDATE New_table AS N
SET kwota  = (
    SELECT A.kwota
    FROM (
        SELECT id_klienta, sum(tzk.ilosc*tk.cena) kwota
        FROM t_zamowienie JOIN t_z_ksiazka tzk USING(id_zam)
             JOIN t_ksiazka tk USING (ISBN)
        WHERE zrealizowane = 1 AND id_klienta = N.id_klienta
        GROUP BY id_klienta ) AS A
);
```

Następnie uaktualniamy kwotę równą NULL w tabeli New_table do średniej z kwot różnych od NULL z tej tabeli.

```
UPDATE New_table
SET kwota = ( SELECT ROUND(AVG(kwota),2)
              FROM New_table
              WHERE kwota IS NOT NULL )
WHERE kwota IS NULL;
```

6.5. Zadania do samodzielnego rozwiązania

W każdym z poniższych zapytań należy wykorzystać tylko podzapytania.

1. Wyświetlić tytuły książek, których autor ma na imię 'Ryszard'.
2. Wyświetlić nazwy wydawców, których książek nie ma w księgarni.
3. Wyświetlić ISBN, tytuł, imię i nazwisko autora tych książek, których nie dostarczył ani 'Konik', ani 'UPS'. Wyniki uszeregować zgodnie z malejącym porządkiem alfabetycznym nazwisk autorów.
4. Wyświetlić powtarzające się nazwiska autorów. Wykorzystać zapytanie skorelowane.
5. Spośród wszystkich książek wyświetlić te, których cena jest największa oraz te, których cena jest druga co do wielkości.
6. Obliczyć i wypisać maksymalną cenę książek wydanych w dowolnym roku, w którym wydano książki zamówione przez klienta o imieniu 'Jan' i nazwisku 'Kowalski'.
7. Wyświetlić tych autorów, których nazwiska zaczynają się od pierwszej litery imienia klienta o identyfikatorze 1 lub których książki kosztują więcej niż cena dowolnej książki napisanej przez Cypriana Norwida i zostały zamówione po '2004-01-10'.
8. Wyświetlić połączone wartości kolumn imię i nazwisko tych autorów, w których nazwisku znajduje się na dowolnym miejscu taka sama litera, jak pierwsza litera nazwiska autora o identyfikatorze 16.
9. Wyświetlić id_klienta, imię i nazwisko tych klientów, którzy zamówili książki wydane w tym samym roku co rok wydania książki o tytule 'Tosca'.
10. Wyświetlić id_autora, imię i nazwisko tych autorów, którzy wydali książki w roku, w którym została wydana więcej niż jedna książka.
11. Wyświetlić ile jest książek, które nigdy nie zostały sprzedane (zamówienie nie zostało jeszcze zrealizowane lub nigdy ich nikt nie zamówił).
12. Wyświetlić najpóźniejszą datę zamówienia książek, zamówionych przez klienta o nazwisku 'Kowalski'.
13. Wyświetlić ISBN, tytuł, imię i nazwisko autora tych książek, które zostały wydane przez 'PIW', 'Znak' lub 'Helion'.
14. Wyświetlić identyfikatory zrealizowanych zamówień o największej wartości.

15. Wyświetlić najmniejszy identyfikator zrealizowanego zamówienia o największej wartości.
16. Wyświetlić `id_klienta`, `imię` i `nazwisko` tych klientów z województwa podlaskiego, którzy zamówili więcej niż jedną książkę.
17. Wyświetlić klientów, którzy zakupili książki tych wydawców, którzy wydali ich więcej niż dwie.
18. Wyświetlić ISBN, tytuł książek oraz nazwiska klientów, którzy zakupili książki autorów o imieniu zaczynającym się na literę 'K' lub 'W'.
19. Wyświetlić dane o autorach tych książek, które nigdy nie zostały sprzedane.
20. Wyszukaj nazwę dostawców tych książek, których wydawca wydał w dowolnym roku więcej niż dwie książki.
21. Wyszukać, ile książek zostało sprzedanych (wziąć pod uwagę tylko zrealizowane zamówienia) w poszczególnych latach, o ile liczba ta mieści się w przedziale od 2 do 4.
22. Zwiększyć kwotę tym klientom z tabeli `New_table`, którzy nie przekroczyli 178 złotych, o średnią z całkowitego kosztu nie zrealizowanych jeszcze zamówień klientów z tabeli `New_table`.
23. Utworzyć tabelę `OpisKsiazki` zawierającą dane na temat książek z następujących kolumn:
    - `ISBN (t_ksiazka)`,
    - `TYTUL (t_ksiazka)`,
    - `NAZWISKO (t_autor)`,
    - `NAZWA (t_wydawca) as wydawca`,
    - `NAZWA (t_dostawca) as dostawca`,

    które opisują książki sprzed 2000 roku.
24. Do tabeli `OpiszKsiazki` dodać kolumnę `cena` typu `numeric(6,2)`.
25. Zmodyfikować dla każdego rekordu w tabeli `OpisKsiazki` dane w kolumnie `cena`, uaktualniając je do rzeczywistej wartości z tabeli `t_ksiazka`.

Rozdział 7

Tworzenie i wykorzystanie perspektyw

Perspektywy tworzymy, aby móc ponownie skorzystać z tworzonych przez nas instrukcji SQL, szczególnie tych bardzo długich i skomplikowanych, bez potrzeby ponownego ich wpisywania. Uważa się je za zapisane zapytania, które pozwalają na utworzenie obiektów bazy danych działających podobnie do tabel, ale posiadających zawartość, która zmienia się dynamicznie i dotyczy bezpośrednio tylko wybranych zgodnie z definicją wierszy. Perspektywy, począwszy od tych dotyczących prostych zapytań do pojedynczej tabeli, aż do zapytań skomplikowanych, dotyczących wielu tabel, oferują dużą elastyczność. W tym rozdziale zostanie omówione tworzenie i praktyczne wykorzystanie perspektyw.

7.1. Podstawowe operacje związane z perspektywami

Polecenie `CREATE VIEW` pozwala na tworzenie widoków, czyli logicznych okien zawierających widok na jedną lub więcej tabel. Widoki są obiektami podobnymi zarówno do formularzy, gdyż nie przechowują danych, jak i do tabel, gdyż możliwe jest użycie tych samych poleceń SQL, które można stosować do tabel. Do wykonania określonych instrukcji SQL niezbędne jest spełnienie zadanych wymagań. Dotyczy to w szczególności polecenia `SELECT`, użytego dla tworzenia widoku i mogącego zawierać klauzulę `GROUP BY` i `DISTINCT`.

Widoki są używane w celu zwiększenia poziomu bezpieczeństwa tabel poprzez ograniczenie dostępu tylko do wybranych kolumn tabeli podstawowej oraz pozwalają ukryć przed użytkownikiem złożoność danych. Bezpieczne jest wybieranie danych za pomocą widoków, ale nie ich modyfikowanie. Nie mogą być modyfikowane dane, które bazują na innych widokach, wielu tabelach, jak i te w których użyto klauzuli `GROUP BY`, `DISTINCT`,

LIMIT, UNION, HAVING oraz funkcji agregujących (SUM(), MIN(), MAX(), COUNT()). Jeżeli widok został utworzony w taki sposób, to wtedy trzeba modyfikować dane w tabelach oryginalnych.

Składnia polecenia tworzenia perspektywy ma postać:

```
CREATE [OR REPLACE]
[TEMP | TEMPORARY] [RECURSIVE] VIEW name [(column_name [,...])]
[WITH (view_option_name [= view_option_value] [,...])]
AS query [WITH [CASCADED | LOCAL] CHECK OPTION];
```

- Klauzula OR REPLACE w składni polecenia CREATE VIEW jest opcjonalna i jej dodanie powoduje, że w przypadku istnienia perspektywy o takiej nazwie, zostanie ona nadpisana. Inną metodą zmiany zapytania utworzonego widoku jest instrukcja ALTER VIEW.
- Możemy tworzyć perspektywę o nazwie name, nadając jej kolumnom nazwy wymienione na liście ujętej w nawiasy okrągłe, która opcjonalnie może być tymczasowa lub rekursywna.
- [WITH (view_option_name [= view_option_value][,...])] określa opcjonalne parametry widoku, takie jak: check_option(enum), security_barrier(boolean), czy security_invoker(boolean).
- Zapytanie query w tworzonej perspektywie określa strukturę oraz zbiór danych. Może mieć niemal dowolną składnię i np. zawierać złączenia na tabelach, przecięcia i podzapytania, a także odnosić się do innych widoków. Zapytanie nie może:
  - zawierać podzapytań w klauzuli FROM,
  - korzystać ze zmiennych systemowych i tych definiowanych przez użytkownika,
  - tabele użyte w zapytaniu muszą istnieć w momencie tworzenia widoku, nie można się odwoływać do tabel tymczasowych.

  Dozwolone jest używanie klauzuli ORDER BY, jednak jest ono ignorowane, jeśli w zapytaniu do widoku również pojawia się ta klauzula.
- Klauzula WITH [CASCADED | LOCAL] CHECK OPTION - ogólnie sprawdza w momencie użycia instrukcji INSERT i UPDATE, czy jest spełniony warunek klauzuli WHERE.

Przykład tworzenia perspektywy, w której domyślnie nazwy kolumn są takie same jak te, które wchodziły w skład zapytania:

```
CREATE VIEW V1 AS
SELECT isbn, tytul, ilosc
FROM t_ksiazka
WHERE ilosc > 3
WITH CHECK OPTION;
```

Przykład tworzenia perspektywy, w której nazwy źródłowych kolumn zostały zmienione.

```
CREATE VIEW V2 (kol1, kol2) AS
SELECT oprawa, count(*) as Count
FROM t_ksiazka
GROUP BY oprawa;
```

Przy zmianie nazw źródłowych kolumn należy pamiętać, że liczba zdefiniowanych kolumn przez widok musi odpowiadać liczbie kolumn zwracanych przez zapytanie `SELECT`.

Aby zobaczyć nazwy utworzonych perspektyw należy wykonać w kliencie `psql` polecenie `\dv` lub `\d+`, a po nich wymienić nazwę widoku.

`ALTER VIEW` dostarcza możliwości modyfikacji definicji perspektywy i ma następującą składnię:

```
ALTER VIEW [IF EXISTS] name
ALTER [COLUMN] column_name SET DEFAULT expression

ALTER VIEW [IF EXISTS] name
ALTER [COLUMN] column_name DROP DEFAULT

ALTER VIEW [IF EXISTS] name
OWNER TO {new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER}

ALTER VIEW [IF EXISTS] name
RENAME [COLUMN] column_name TO new_column_name

ALTER VIEW [IF EXISTS] name
RENAME TO new_name

ALTER VIEW [IF EXISTS] name
SET SCHEMA new_schema

ALTER VIEW [IF EXISTS] name
SET (view_option_name [= view_option_value] [,... ])

ALTER VIEW [IF EXISTS] name
RESET (view_option_name [,... ])
```

Do perspektywy odwołujemy się tak samo jakby była zwykłą tabelą.

```
SELECT column_name [,...]
FROM nazwa_widoku;
```

Przykładowo dla perspektyw `V1` i `V2` zdefiniowanych wcześniej, wyświetlenie ich zawartości odbywać się będzie w następujący sposób:

```
SELECT tytul FROM V1;
SELECT * FROM V2;
```

Aby utworzyć perspektywę, niezbędne jest posiadanie przez użytkownika uprawnień `CREATE VIEW` (również dla pozostałych instrukcji działających na perspektywie) w bazie danych i `SELECT` dla wybranej tabeli. Należy również pamiętać, że modyfikowanie zbioru danych (oryginalnych danych) ma wpływ na zawartość danych w perspektywie.

Usuwanie perspektywy:

```
DROP VIEW [IF EXISTS] nazwa_widoku;
```

Klauzula `[IF EXISTS]` jest opcjonalna i ma na celu sprawdzenie, czy perspektywa, którą chcemy usunąć na pewno istnieje. Jeśli nie istnieje instrukcja zostanie zignorowana, o czym zostaniemy poinformowani odpowiednim komunikatem. Przykładowo:

```
DROP VIEW IF EXISTS V1;
```

usuwa perspektywę o nazwie `V1` o ile została wcześniej utworzona i nadal istnieje w bazie danych.

7.2. Perspektywy w działaniu

Rozróżnia się dwa rodzaje perspektyw:

- prostą, która udostępnia dane z pojedynczej tabeli, a w jej definicji nie stosuje się:
  - operacji na zbiorach,
  - funkcji,
  - grupowania wierszy;
- złożoną, która udostępnia dane z wielu tabel oraz operacje na zbiorach, łączenie tabel, funkcje i grupowanie wierszy; wówczas do perspektywy można kierować tylko polecenie `SELECT`.

Zdefiniujemy perspektywę w oparciu o zapytanie, które ma wyznaczyć liczbę dostępnych w księgarni książek dla każdego wydawcy, grupując książki według nazw wydawców, oraz najniższą cenę w każdej grupie.

```
CREATE VIEW ksiazka_wydawca(nazwa_wydawcy, Liczba, min) AS
SELECT nazwa, count(1), min(cena)
FROM t_ksiazka INNER JOIN t_wydawca ON (wydawca = id_wydawcy)
GROUP BY nazwa
ORDER BY 1 DESC;
```

Wyświetlamy wyznaczone dane, odwołując się do perspektywy.

```
SELECT * FROM ksiazka_wydawca;
```

```
SELECT *
FROM ksiazka_wydawca
ORDER BY min DESC LIMIT 3;
```

Drugi przykład pokazuje, że wyniki perspektywy można posortować inaczej niż zrobiono to w definicji perspektywy.

Widok z możliwością uaktualniania pozwala na uaktualnienie tabel podstawowych, które go tworzą. Dopóki spełnione są określone warunki, dopóty można wykonywać polecenia `UPDATE`, `DELETE`, a nawet `INSERT` w widoku jak w zwykłej tabeli. Widok nie ma możliwości uaktualniania, jeżeli zawiera między innymi klauzule grupowania, sumy, czy funkcję agregującą. Zapytanie, które modyfikuje dane może zawierać złączenie, ale wszystkie zmieniane kolumny muszą znajdować się w jednej tabeli.

```
CREATE VIEW V3 AS
SELECT id_wydawcy, nazwa
FROM t_wydawca
WHERE nazwa ilike '%a%';
```

```
UPDATE v3
SET nazwa = upper(nazwa)
WHERE id_wydawcy = 3;
```

```
SELECT nazwa FROM v3 WHERE id_wydawcy = 3;
```

```
SELECT nazwa FROM t_wydawca WHERE id_wydawcy = 3;
```

```
INSERT INTO v3
VALUES (1000, 'Kongo');
```

Po wstawieniu nowego rekordu poprzez perspektywę, próba odwołania się, również poprzez perspektywę do rekordu o identyfikatorze wydawcy równym 1000, kończy się niepowodzeniem i pustym zbiorem wynikowym. Dzieje się tak, dlatego że nazwa wydawcy nie spełnia warunku w klauzuli `WHERE` definicji perspektywy.

```
SELECT nazwa FROM v3 WHERE id_wydawcy = 1000;
 nazwa
-------
(0 rows)
```

Gdy tymczasem odwołanie do tabeli źródłowej pokazuje wstawione dane.

```
SELECT nazwa FROM t_wydawca WHERE id_wydawcy = 1000;
 nazwa
-------
 Kongo
(1 row)
```

Ważne, aby przy wstawianiu danych poprzez perspektywę zadbać o uzupełnienie wszystkich obowiązkowych kolumn w tabeli, do której perspektywa się odwołuje lub zdefiniować dla tych kolumn wartość domyślną.

Teraz poprzez perspektywę usuwamy wiersze.

```
DELETE FROM v3
WHERE id_wydawcy = 3;
```

Sprawdzamy, czy istnieje wydawca o identyfikatorze 3.

```
SELECT nazwa FROM v3 WHERE id_wydawcy = 3;
 nazwa
-------
(0 rows)
```

Również w źródłowej tabeli nie znajdziemy usuniętego rekordu.

```
SELECT nazwa FROM t_wydawca WHERE id_wydawcy = 3;
 nazwa
-------
(0 rows)
```

Przykłady perspektyw opartych na łączeniu tabel.

```
CREATE VIEW v4 AS
SELECT ISBN, tytul, nazwa
FROM t_ksiazka INNER JOIN t_wydawca ON wydawca = id_wydawcy
WHERE isbn < 50;

SELECT * FROM v4 WHERE ISBN = 45;

CREATE VIEW autor_ksiazka
AS
SELECT imie, nazwisko, tytul, rok
FROM t_autor INNER JOIN t_a_ksiazki USING (id_autora)
     INNER JOIN t_ksiazka USING (ISBN)
WHERE rok > '1997';
```

```
SELECT * FROM autor_ksiazka;

CREATE VIEW ksiazka_statystyka(rok, cena_min,cena_max,cena_sr)
AS
SELECT rok, min(cena), max(cena), round(avg(cena),2)
FROM t_ksiazka
GROUP BY rok;

SELECT *
FROM ksiazka_statystyka
WHERE rok LIKE '_9%'
ORDER BY 1;
```

7.3. Zadania do samodzielnego rozwiązania

Do każdego z poniższych zadań zdefiniować perspektywy, których nazwy składają się ze słowa `zad_nr`, gdzie `nr` jest numerem rozwiązywanego zadania. Wyświetlić dane z każdej perspektywy.

1. Wyświetlić tytuły książek i identyfikator autora tych książek, które zostały zamówione w ilości nie mniejszej niż 2.
2. Przyjmując, że autor (lub potomek autora) dostaje 2% od sprzedanego egzemplarza, wyświetlić kwoty zarobionych pieniędzy ze sprzedaży (wziąć pod uwagę tylko zrealizowane zamówienia).
3. Wyświetlić informacje o książkach, których cena nie jest większa niż ceny książek dostarczanych przez dostawcę z siedzibą w miejscowości zawierającej literę 's' na dowolnym miejscu.
4. Wyświetlić tytuły tych książek, których ilość sprzedanych egzemplarzy przewyższa ilość posiadanych egzemplarzy (wykorzystać zapytanie skorelowane).
5. Wyświetlić tytuły książek i identyfikator autora tych książek, które zostały sprzedane w ilości mniejszej niż 2 (wykorzystać podzapytanie skorelowane).
6. Wyświetlić identyfikatory wydawców, którzy wydali książki co najmniej 3 różnych autorów.
7. Wyznaczyć autorów, którzy wydali książki w roku, w którym nikt inny nic nie wydał. (Użyć operatora `NOT EXISTS`).

Czytelnika chcącego udoskonalić swoje umiejętności pracy z perspektywami zachęcamy do zdefiniowania perspektyw w oparciu o zadania do samodzielnego rozwiązania proponowane w innych rozdziałach.

Rozdział 8

Tworzenie i wykorzystanie indeksów

Zwiększenie wydajności zapytań w bazach danych uzyskujemy, stosując indeksy. Indeks bazy danych to odrębny plik zawierający wstępnie przygotowany i uporządkowany zbiór (lub podzbiór) odsyłaczy do danych zgodny z określonymi warunkami. Gdy podczas wykonywania zapytania dostępny jest indeks zawierający interesujące nas dane, planer, używając różnych mechanizmów działających na serwerze do analizy żądania i wyboru najlepszego pod względem efektywności czasowej i pamięciowej sposobu jego wykonania, może wykorzystać wstępnie przygotowane i uporządkowane dane z tego indeksu. Jeśli indeks jest niedostępny, baza danych musi wielokrotnie skanować wszystkie rekordy, by wybrać te z nich, które zawierają interesujące nas dane. Nawet jeśli wszystkie informacje znajdują się na początku przeszukiwanego zbioru danych, to muszą być skanowane wszystkie rekordy.

O przydatności danego indeksu decyduje jego klucz. Klucz indeksu to kolumna lub grupa kolumn, do których indeks jest stosowany. Indeks może przyspieszyć zapytanie tylko wtedy, gdy jego klucz zawiera jedną lub więcej kolumn użytych w predykacie zapytania, tj. w klauzuli `WHERE` (ważna jest również kolejność kolumn). Należy podkreślić, że w relacyjnych baza danych zawsze w tworzonej tabeli definiowany jest indeks na jej kluczu głównym. Pozostałe indeksy definiuje użytkownik i powinien to robić z rozwagą, zdając sobie sprawę, że każdy indeks zwiększa zasoby bazy danych, a jednocześnie, gdy źle zdefiniowane zostaną kryteria wyboru w zapytaniu, nigdy może nie być wykorzystywany.

W celu zwiększenia wydajności wyszukiwania PostgreSQL stosuje wiele różnych mechanizmów indeksowania, takich jak: B-tree, indeksy z haszowaniem, indeksy GIN (ang. *Generalized Inverted Index*) czy indeksy GiST (ang. *Generalized Search Tree*). Ponieważ najczęściej używanym typem indeksu jest indeks typu B-tree, w tym rozdziale przedsta-

wimy tworzenie i wykorzystanie tego typu indeksów na przykładzie prostej tabeli, ale z dużą liczbą wierszy. Wyjaśnimy, dlaczego są one potrzebne, a także jakie wady i zalety posiadają.

8.1. Indeks typu B-tree

Indeksy B-tree to specyficzny rodzaj indeksu bazy danych, porządkujący określone zakresy wartości kluczy w postaci struktury drzewiastej, tzw. drzewa zrównoważonego. Bez zagłębiania się w teorię struktury B-tree i algorytmów przechodzenia po drzewie, bo zarządza nimi sama baza, postaramy się w prosty sposób wyjaśnić zasadę działania indeksów tego typu.

Indeks B-tree porządkuje wiersze zgodnie z wartościami klucza i dzieli uporządkowaną listę na zakresy. Zakresy te są zorganizowane w strukturę drzewa, w której węzeł główny definiuje szeroki zestaw zakresów, a każdy poziom poniżej zawęża zakresy. Rozważmy przykład księgarni, w której książki ułożone są według nazwiska autora. Załóżmy, że szukamy książki autorstwa Sienkiewicza - tu imię nie ma znaczenia, bo porządek wyznaczony jest według nazwisk autorów. W księgarni znajdują się rzędy regałów, przy czym każdy rząd oznaczony jest początkowymi literami nazwisk autorów, na przykład `A-F`, dalej `G-O`, potem `P-Z`. Przechodzimy do konkretnego regału o zakresie wartości kluczy `P-Z` i widzimy, że każda półka posiada tabliczkę z mniejszym zakresem nazwisk autorów, np. `PA-PZ`, potem `SA-SZ` i tak dalej. Mamy więc tutaj do czynienia z podzakresami większych zakresów. Z kolei książki na każdej półce ułożone są alfabetycznie wg. nazwisk autorów, dzięki czemu możemy szybko znaleźć książkę Sienkiewicza.

Organizacja indeksu B-tree odzwierciedla opisany wyżej przykład dotyczący księgarni. Oczywiście w wypadku indeksu bazodanowego mamy do czynienia z większą ilością zakresów, jednak idea bazowa pozostaje ta sama. Każdy węzeł w B-tree nazywany jest blokiem. Bloki na każdym poziomie drzewa, z wyjątkiem ostatniego, nazywane są blokami gałęzi, a te na ostatnim poziomie nazywane są blokami liści. Wpisy w blokach gałęzi składają się z identyfikatora zakresu i wskaźnika do bloku na następny poziom drzewa. Każdy wpis w bloku liścia reprezentuje wiersz i składa się z wartości klucza (np. „Sienkiewicz”) i identyfikatora wiersza niezbędnego do uzyskania danych zawartych w tym wierszu.

Aby utworzyć indeks określonego typu (np. `BTREE, HASH, GIST` itp.) dla zbioru danych używamy instrukcji `CREATE INDEX` wraz z wymienieniem na liście kolumn tych, na których indeks w obrębie podanej tabeli jest nakładany. Ponadto możemy nałożyć dodatkowe warunki i ograniczenia, aby indeks był bardziej selektywny.

```
CREATE INDEX index_name
ON table_name [USING index_type](col[,...])
[WHERE condition];
```

Podczas wykonywania polecenia `CREATE INDEX` indeks B-Tree jest tworzony domyślnie, działa na wszystkich typach danych i może być używany do pobierania wartości NULL.

8.2. Funkcje składowane

W tym podrozdziale krótko omówimy tworzenie i wykorzystanie funkcji składowanych do opracowywania zapytań i wielokrotnego ich wykorzystania, w tym do utworzenia i wypełnienia danymi przykładowej tabeli, na której pokażemy działanie indeksów.

Funkcje w SQL-u to wyodrębnione bloki kodu, które składowane są w bazie danych. Umożliwiają wydajne wielokrotne wykorzystanie kodu (składowanie na serwerze już skompilowanych funkcji wymaga tylko ich uruchomienia) do powtarzania i modyfikowania instrukcji i zapytań. Jednak najważniejszą zaletą funkcji jest to, że pozwalają na podział kodu na mniejsze, możliwe do testowania porcje.

Do tworzenia funkcji składowanych służy polecenie:

```
CREATE [OR REPLACE] FUNCTION function_name ([arg[,...]])
RETURNS returning_value_type AS $$
[DECLARE var_name var_type;]
BEGIN
  body_function;
  [RETURN value;]
END; $$
LANGUAGE plpgsql;
```

Poszczególne elementy funkcji:

- `function_name` - nazwa przypisana do funkcji i służąca do jej późniejszego uruchamiania;
- `[arg[,...]]` - opcjonalna lista argumentów funkcji. Może być pusta lub zawierać listę wartości różnych typów danych (literałów) albo listę nazwanych argumentów;
- `returning_value_type` - typ danych zwracany przez funkcję;
- opcjonalna instrukcja `DECLARE` to blok deklaracji zmiennych lokalnych funkcji;
- `body_function` - ciało funkcji;
- `[RETURN value;]` - opcjonalnie, wartość zwracana przez funkcję;
- `plpgsql` określa język używany w funkcji, w przypadku tego podręcznika wykorzystujemy tylko ten.

Zdefiniujemy dla przykładu funkcję `count_order`, która nie przyjmuje żadnych argumentów i zwraca wartość całkowitą, która określa liczbę złożonych i zrealizowanych zamówień.

```
CREATE FUNCTION count_order()
RETURNS integer AS $$
DECLARE
  total integer;
BEGIN
  SELECT COUNT(id_zam) INTO total
  FROM t_zamowienie
  WHERE zrealizowane = 1;
  RETURN total;
END;
$$ LANGUAGE plpgsql;
```

W ciele funkcji użyliśmy instrukcji `SELECT..INTO...`, która przekierowuje wynik zapytania do zmiennej zadeklarowanej w bloku `DECLARE` i zwraca ją. Zdefiniowaną funkcję możemy używać tam, gdzie jest taka potrzeba, jak każdej wbudowanej, standardowej funkcji PostgreSQL.

Wywołanie funkcji składowanej `count_order` w najprostszej postaci wygląda w następujący sposób: `SELECT count_order();`

Zdefiniujemy teraz funkcję `max_order`, która umożliwi obliczenie i zwrócenie największej wartości zamówienia spośród już zrealizowanych.

```
CREATE FUNCTION max_order()
RETURNS numeric AS $$
DECLARE
  maximum numeric;
BEGIN
  WITH A AS (
     SELECT id_zam, sum(tk.cena * tzk.ilosc) as total
     FROM t_ksiazka tk JOIN t_z_ksiazka tzk USING (ISBN)
          JOIN t_zamowienie USING (id_zam)
     WHERE zrealizowane = 1
     GROUP by tzk.id_zam
  )
  SELECT max(A.total) INTO maximum
  FROM A;
  RETURN maximum;
END;
$$ LANGUAGE plpgsql;
```

Zdefiniowaliśmy funkcję, po uruchomieniu której poleceniem `SELECT max_order()` możemy uzyskać informację o aktualnej, najwyższej kwocie zrealizowanego zamówienia.

Utworzymy funkcję przyjmującą argumenty, która umożliwia obliczenie sumy cen książek wyszukanych zgodnie z zadanymi parametrami (argumentami).

```
CREATE FUNCTION search_sum_book(year char(4), t char(1))
RETURNS numeric AS $$
DECLARE
  result numeric;
BEGIN
  SELECT sum(cena) INTO result
  FROM t_ksiazka
  WHERE rok = year AND tytul ilike '%'||t||'%';
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

A następnie wywołamy ją, by obliczyć sumę cen książek z 1997 roku, w których tytule na dowolnym miejscu znajduje się litera 't' (bez rozróżniania wielkości liter).

```
SELECT search_sum_book('1997', 't');
```

W kliencie `psql` używając polecenia `\df` możemy pobrać listę funkcji składowanych, jej zmiennych, typów danych argumentów i typu zwracanej wartości.

```
postgres=# \df
                                List of functions
 Schema |      Name       | Result    |     Argument     | Type
        |                 | data type |    data types    |
--------+-----------------+-----------+------------------+------
 public | count_order     | integer   |                  | func
 public | indeksy         | integer   |                  | func
 public | max_order       | numeric   |                  | func
 public | search_sum_book | numeric   | year character,  | func
        |                 |           | t character      |
(4 rows)
```

Z kolei poleceniem `\sf function_name` możemy wyświetlić definicję już istniejącej funkcji, np. `postgres=# \sf max_order`.

W ciele funkcji możemy również używać instrukcji iteracyjnych, np. `FOR`. Jej zastosowanie pokażemy na prostym przykładzie obliczającym sumę `n` początkowych liczb całkowitych dodatnich, gdzie `n` jest argumentem funkcji.

```
-- definicja funkcji PL/PGSQL
CREATE FUNCTION suma(n int)
RETURNS int AS $$
DECLARE   -- blok deklaracji zmiennych
   i int;
   suma int;
BEGIN
   suma := 0;  -- przypisanie wartości początkowej
   FOR i IN 1..n LOOP  -- instrukcja iteracyjna FOR
     suma := suma + i;
   END LOOP;
   RETURN suma;
END;
$$ LANGUAGE plpgsql;

SELECT suma(4); -- wywołanie funkcji wypisze na ekranie 10
```

Szczegółowe informacje o funkcjach i procedurach składowanych zainteresowany czytelnik odnajdzie w dokumentacji PostgreSQL.

8.3. Przykładowa sesja

W celu odczytania czasu wykonania poleceń SQL można wykorzystać włączenie lub wyłączanie wyświetlania czasu trwania każdej instrukcji SQL poleceniem `\timing [ON/OFF]` – domyślnie `OFF` lub odczytać `Execution Time` z informacji wyświetlanych przez polecenia `EXPLAIN` lub `EXPLAIN ANALYZE`.

Uruchomienie polecenia `EXPLAIN` lub `EXPLAIN ANALYZE` nie wykona zapytania i nie zwróci wartości. Zamiast tego zwróci opis wraz z kosztami przetwarzania poszczególnych etapów planu, a `ANALYZE` doda jeszcze czas planowania i wykonania zapytania.

```
EXPLAIN SELECT * FROM t_ksiazka;
                         QUERY PLAN
---------------------------------------------------------------
 Seq Scan on t_ksiazka  (cost=0.00..14.20 rows=420 width=166)
(1 row)
```

Plan wykonania zapytania zawiera informacje o typie skanowania używanego w zapytaniu. Tu `Seq Scan` oznacza skanowanie sekwencyjne, liniowy sposób przeszukiwania pełnego zbioru danych, w którym sprawdzany jest każdy rekord w tabeli i porównywany

z kryteriami skanowania sekwencyjnego w celu wyboru tych z nich, które spełniają kryterium określone w klauzuli `WHERE`. Skanowanie sekwencyjne jest najczęściej stosowane, gdy tabela jest mała lub pole używane w poszukiwaniu zawiera wiele powtarzających się wartości, lub planer stwierdza, że skanowanie sekwencyjne przy podanych kryteriach jest równe (lub bardziej) wydajne niż inne metody skanowania.

Dodatkowo otrzymujemy informacje o kosztach operacji przygotowawczych, np. posortowanie danych (`cost=0.00`). Miara ta podawana jest w jednostkach kosztu, a nie sekundach i często jest to liczba żądań kierowanych do dysku lub pobrań stron. Następnie mamy podaną wartość (`14.20`) określającą łączny koszt wykonania zapytania, jeśli pobrane zostaną wszystkie wiersze (nie zawsze tak jest). Następna wartość (`rows=420`) w planie określa łączną liczbę wierszy dostępnych do zwrócenia, jeśli plan wykonany zostanie w całości. A ostatnia (`width=166`), to długość każdego wiersza w bajtach.

W rozważanym w tym rozdziale przykładzie w celu porównywania czasów wykonania zapytań używać będziemy polecenia `EXPLAIN ANALYZE`.

Tworzymy tabelę testową i wstawiamy do niej 1.000.000 przykładowych rekordów. Do pliku o nazwie `indeksy.sql` zapisz definicję funkcji bezargumentowej o nazwie `indeksy`, w której trzy zagnieżdżone instrukcje iteracyjne `FOR` generują wartości wstawiane do tabeli testowej.

```
-- definicja funkcji PL/PGSQL
CREATE FUNCTION indeksy()
RETURNS int AS $$
DECLARE
   i int;
   j int;
   k int;
BEGIN
   DROP TABLE IF EXISTS indeksy;

   CREATE TABLE indeksy (
    id SERIAL PRIMARY KEY,
    x INT,
    y INT,
    z INT
   );
   FOR i IN 1..100 LOOP  -- instrukcja iteracyjna FOR
     FOR j in 1..100 LOOP
       FOR k in 1..100 LOOP
         INSERT INTO indeksy (x,y,z) VALUES(i,j,k);
```

```
      END LOOP;
    END LOOP;
  END LOOP;
  RETURN 0;
END;
$$ LANGUAGE plpgsql;

SELECT indeksy(); -- wywolanie funkcji
```

Sprawdzamy liczbę wierszy w tabeli `indeksy`.

```
postgres=# SELECT COUNT(*) FROM indeksy;
+----------+
| count(*) |
+----------+
| 1000000 |
+----------+
(1 row)
```

Sprawdzamy, jakie indeksy mamy w tej chwili założone na tabeli `indeksy` – jest tylko indeks dla klucza głównego.

```
postgres=# \d indeksy
 Table "public.indeksy"
 Column |  Type   | Colla- |          |
        |         | tion   | Nullable |              Default
--------+---------+--------+----------+-----------------------------------
 id     | integer |        | not null | nextval('indeksy_id_seq'::regclass)
 x      | integer |        |          |
 y      | integer |        |          |
 z      | integer |        |          |
Indexes:
    "indeksy_pkey" PRIMARY KEY, btree (id)
```

Wykonujemy przykładowe zapytanie.

```
postgres=# SELECT * FROM indeksy WHERE x=1 AND y=10 AND z=100;

+------+------+------+------+
| id   | x    | y    | z    |
+------+------+------+------+
| 1000 |    1 |   10 |  100 |
+------+------+------+------+
(1 row)
```

W planie wykonania zapytania SQL-owego widać, że żaden indeks nie został użyty (bo nie ma na razie żadnych indeksów, które mogłyby pomóc w szybszym wykonaniu zapytania).

```
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=1 AND y=10 AND z=100;
QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..13697.77 rows=1 width=16)
         (actual time=2.827..105.610 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   -> Parallel Seq Scan on indeksy (cost=0.00..12697.67 rows=1 width=16)
                            (actual time=52.130..82.390 rows=0 loops=3)
         Filter: ((x = 1) AND (y = 10) AND (z = 100))
         Rows Removed by Filter: 333333
 Planning Time: 0.400 ms
 Execution Time: 105.688 ms
(8 rows)
```

Pierwszym wartym uwagi aspektem powyższego zapytania jest to, że obejmuje ono kilka różnych kroków. Zapytanie niższego poziomu wykonuje skanowanie sekwencyjne na podstawie podanych wartości. Skanowanie sekwencyjne jest tu przeprowadzane równolegle (`Parallel Seq Scan`) i planowane jest użycie dwóch wątków roboczych. To, czy serwer PostgreSQL zastosuje skanowanie równoległe zależy zarówno od konfiguracji serwera, jak i możliwości sprzętowych komputera. W tym konkretnym przykładzie PostgreSQL uznał, że skanowanie równoległe może zapewnić wyższą wydajność, dlatego przydziela do tej operacji dwa wątki. Na wyższym poziomie planu widoczny jest krok `Gather`, wykonywany na początku zapytania. Koszt operacji przygotowawczych jest niezerowy (`1000`), a łączny koszt wynosi `13697.77`.

Tworzymy trzy indeksy, po jednym na każdy atrybut niekluczowy (odpowiednio na kolumnach x, y oraz z).

```
postgres=# CREATE INDEX idx_x ON indeksy (x);
CREATE INDEX

postgres=# CREATE INDEX idx_y ON indeksy (y);
CREATE INDEX

postgres=# CREATE INDEX idx_z ON indeksy (z);
CREATE INDEX

postgres=# \d indeksy;
```

```
 Table "public.indeksy"
 Col |  Type   | Colla- |          |
 umn |         | tion   | Nullable |               Default
-----+---------+--------+----------+-------------------------------------
 id  | integer |        | not null | nextval('indeksy_id_seq'::regclass)
 x   | integer |        |          |
 y   | integer |        |          |
 z   | integer |        |          |
Indexes:
    "indeksy_pkey" PRIMARY KEY, btree (id)
    "idx_x" btree (x)
    "idx_y" btree (y)
    "idx_z" btree (z)
```

Wykonujemy ponownie to samo zapytanie i sprawdzamy w planie wykonania zapytania (`EXPLAIN ANALYZE`), czy system użył zdefiniowane wcześniej indeksy.

```
postgres=# SELECT * FROM indeksy WHERE x=1 AND y=10 AND z=100;
  id  | x | y  |  z
------+---+----+-----
 1000 | 1 | 10 | 100
(1 row)

postgres=# EXPLAIN ANALYZE SELECT *
postgres-#FROM indeksy WHERE x=1 AND y=10 AND z=100;
                            QUERY PLAN
------------------------------------------------------------------------
 Index Scan using idx_x on indeksy (cost=0.42..409.68 rows=1 width=16)
                           (actual time=2.095..12.810 rows=1 loops=1)
   Index Cond: (x = 1)
   Filter: ((y = 10) AND (z = 100))
   Rows Removed by Filter: 9999
 Planning Time: 0.449 ms
 Execution Time: 12.893 ms
(6 rows)
```

Tworzymy indeks złożony, zbudowany na trzech kolumnach.

```
postgres=# CREATE INDEX idx_xyz ON indeksy (x, y, z);
CREATE INDEX
```

Ponownie wyświetlamy informacje o indeksach zdefiniowanych w tabeli `indeksy`.

```
postgres=# \d indeksy;
                          Table "public.indeksy"
 Col- |  Type   | Colla- | Nullable |               Default
 umn  |         | tion   |          |
------+---------+--------+----------+-------------------------------------
 id   | integer |        | not null | nextval('indeksy_id_seq'::regclass)
 x    | integer |        |          |
 y    | integer |        |          |
 z    | integer |        |          |
Indexes:
    "indeksy_pkey" PRIMARY KEY, btree (id)
    "idx_x" btree (x)
    "idx_xyz" btree (x, y, z)
    "idx_y" btree (y)
    "idx_z" btree (z)
```

W poleceniu `EXPLAIN ANALYZE` widać, że tym razem system użył jednego potrójnego indeksu, zamiast trzech wcześniej zdefiniowanych indeksów pojedynczych.

```
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=1 AND y=10 AND z=100;
                                QUERY PLAN
-------------------------------------------------------------------------
Index Scan using idx_xyz on indeksy (cost=0.42..8.45 rows=1 width=16)
                            (actual time=0.170..0.178 rows=1 loops=1)
   Index Cond: ((x = 1) AND (y = 10) AND (z = 100))
 Planning Time: 1.514 ms
 Execution Time: 0.273 ms
(4 rows)
```

Usuwamy teraz indeksy pojedyncze:

```
postgres=# DROP INDEX idx_x;
DROP INDEX

postgres=# DROP INDEX idx_y;
DROP INDEX
```

```
postgres=# DROP INDEX idx_z;
DROP INDEX
```

Pozostał tylko indeks złożony (na trzech kolumnach: x, y, z) i indeks ten zostanie użyty tylko wówczas, gdy klauzula `WHERE` będzie miała odpowiednią postać:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*)
postgres-# FROM indeksy WHERE x=1 AND y=10 AND z=100;
                                QUERY PLAN
-------------------------------------------------------------------------
Aggregate (cost=8.45..8.46 rows=1 width=8)
          (actual time=0.232..0.235 rows=1 loops=1)
-> Index Only Scan using idx_xyz on indeksy (cost=0.42..8.45 rows=1 width=0)
                            (actual time=0.195..0.202 rows=1 loops=1)
         Index Cond: ((x = 1) AND (y = 10) AND (z = 100))
         Heap Fetches: 1
 Planning Time: 0.880 ms
 Execution Time: 0.379 ms
(6 rows)

postgres=# EXPLAIN ANALYZE SELECT COUNT(*)
postgres-# FROM indeksy WHERE z=100;
                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate  (cost=11625.28..11625.29 rows=1 width=8)
                     (actual time=102.133..114.030 rows=1 loops=1)
   ->  Gather  (cost=11625.06..11625.27 rows=2 width=8)
               (actual time=101.974..114.014 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (cost=10625.06..10625.07 rows=1 width=8)
                          (actual time=87.578..87.579 rows=1 loops=3)
               ->  Parallel Seq Scan on indeksy
                   (cost=0.00..10614.33 rows=4292 width=0)
                   (actual time=0.084..86.934 rows=3333 loops=3)
                     Filter: (z = 100)
                     Rows Removed by Filter: 330000
 Planning Time: 0.454 ms
 Execution Time: 114.184 ms
(10 rows)
```

Porównanie czasów wykonań poleceń:

```
+----+---------+-------------------------------------------------------------+
|ID  | Duration| Query                                                       |
|    |    (ms) |                                                             |
+----+---------+-------------------------------------------------------------+
| 1  |   9.869 | SELECT COUNT(*) FROM indeksy WHERE x=1                      |
| 2  | 111.066 | SELECT COUNT(*) FROM indeksy WHERE y=10                     |
| 3  | 116.288 | SELECT COUNT(*) FROM indeksy WHERE z=100                    |
| 4  | 107.074 | SELECT COUNT(*) FROM indeksy WHERE z=100 AND y=10           |
| 5  |   0.281 | SELECT COUNT(*) FROM indeksy WHERE x=1 AND y=10 AND z=100   |
| 6  |   0.266 | SELECT COUNT(*) FROM indeksy WHERE z=100 AND y=10 AND x=1   |
+----+---------+-------------------------------------------------------------+
```

Przetestujemy wykorzystanie klauzuli `LIMIT` w zapytaniach. Usuniemy istniejący indeks złożony, a następnie zapytamy się o rekordy, o których wiemy, że jeden występuje na początku tabeli, a drugi na końcu. Oczywiście w praktycznych zastosowaniach najczęściej nie znamy fizycznego położenia poszukiwanego rekordu lub rekordów i dlatego też używanie klauzuli `LIMIT` może (zwykle) nie dać tak dobrych wyników. Wykonaj samodzielnie poniższe instrukcje i porównaj plany wykonania zapytań.

```
postgres=# DROP INDEX idx_xyz;
DROP INDEX
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=1 AND y=1 AND z=1;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=100 AND y=100 AND z=100;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=1 AND y=1 AND z=1 LIMIT 1;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indeksy WHERE x=100 AND y=100 AND z=100 LIMIT 1;
```

8.4. Skuteczne korzystanie z indeksów

Indeksy mogą wydawać się oczywistym sposobem na przyspieszenie działania zapytań, ale nie zawsze dają oczekiwany efekt. Rozważmy klika typowych sytuacji:

- Pole, dla którego tworzony jest indeks, często jest modyfikowane. Wstawianie, usuwanie wierszy z tabeli powoduje, że utworzony indeks może stać się niewydajny, gdyż utworzony został na danych, które już nie istnieją lub zmieniły wartość. Wymaga to reorganizacji indeksów. W SQL-u indeksy danych można odtworzyć za pomocą polecenia `REINDEX`, ale wymaga ono przeanalizowania kosztów, technik i strategii.

- Indeks jest nieaktualny, a istniejące referencje są nieprawidłowe, albo występują segmenty niezindeksowanych danych, względem których planer nie może wykorzystać indeksu. PostgreSQL zwykle automatycznie aktualizuje indeksy w reakcji na zmiany w tabeli. Niestety może się zdarzyć sytuacja, gdy aktualizacja automatyczna nie przebiega poprawnie. Wtedy konieczna jest jego „ręczna" aktualizacja.
- Często wyszukiwane są rekordy przy użyciu tych samych kryteriów wyszukiwania dla określonego pola. Można wtedy zamiast stosować indeks na wszystkich danych w podanej kolumnie, ograniczyć go do utworzenia częściowego indeksu z wykorzystaniem podzbioru danych zgodnego z kryterium wyboru. Tworzymy w ten sposób mniejszy, przez to wydajniejszy, indeks, który jest łatwiejszy w utrzymaniu i można stosować go w bardziej skomplikowanych zapytaniach.
- Baza danych nie jest duża. Wówczas koszt tworzenia, utrzymania i stosowania indeksu może przewyższać skanowanie sekwencyjne, które dla małej liczby danych (zwłaszcza jeśli dotyczy to danych składowanych już w pamięci RAM) jest dość szybkie. Zatem utworzenie indeksu, nie gwarantuje nam, że zostanie on użyty w planie wykonania zapytania, przynosząc tylko obciążenie bazy danych.

W rozdziale tym prezentowane były plany wykonania zapytań odnoszących się tylko do jednej tabeli. Wraz ze skomplikowaniem zapytań wynikającym z łączenia tabel, plany ich wykonań również stają się bardziej skomplikowane. Już złączenie dwóch tabel komplikuje plan wykonania zapytania, co wynika z potrzeby nie tylko pobrania większej liczby danych z dysku twardego, ale również dopasowania danych (na podstawie klucza stosowanego w złączeniu) z jednej tabeli do danych z drugiej. Umiejętność interpretowania i rozumienia takich planów wykonania jest bez wątpienia bardzo ważna, ale omawianie tego zagadnienia wykracza poza zakres tego podręcznika. Zainteresowanego czytelnika odsyłamy po dodatkowe informacje do dokumentacji PostgreSQL.

8.5. Zadania do samodzielnego rozwiązania

1. Z poziomu konsoli `psql` PostgreSQL uruchom ponownie skrypt `indeksy.sql`, korzystając np. z polecenia

   `\i ścieżka\nazwa_pliku`.

   Procedura testowa będzie polegała na kilkukrotnym wykonaniu tego samego zapytania, ale dla zdefiniowanych wcześniej odpowiednich indeksów (kolejność tworzenia indeksów znajduje się w tabeli 8.1):

   ```
   SELECT * FROM indeksy WHERE x=1 AND y=10 AND z=100;
   ```

   i odczytaniu czasu odpowiedzi (czas odpowiedzi w milisekundach, z dokładnością do 3 miejsc po przecinku, należy zapisać w trzeciej kolumnie tabeli 8.1). W kolejnych

próbach zmieniamy wyłącznie organizację indeksów założonych na tabeli `indeksy`, bez zmiany wydawanego zapytania. Przykładowo pozycja numer 5 w tabeli 8.1 („Indeksy proste na kolumnach `x`, `y`") oznacza, że w chwili wykonywania polecenia `SELECT` na tabeli testowej założone są tylko dwa indeksy proste, na kolumnach odpowiednio `x` oraz `y`. Aby przejść do pozycji 6, należy usunąć indeks na kolumnie `y`, a założyć indeks na kolumnie `z` itd.

Tabela 8.1: Kolejność tworzenia indeksów w procedurze testowej i czasy wykonania polecenia SELECT

L.P.	Aktualna konfiguracja indeksów	Czas wykonania polecenia `SELECT`
1	Brak indeksów	
2	Indeks prosty na kolumnie `x`	
3	Indeks prosty na kolumnie `y`	
4	Indeks prosty na kolumnie `z`	
5	Indeksy proste na kolumnach `x`, `y`	
6	Indeksy proste na kolumnach `x`, `z`	
7	Indeksy proste na kolumnach `y`, `z`	
8	Indeksy proste na kolumnach `x`, `y`, `z`	
9	Indeks złożony na kolumnach `x`, `y`	
10	Indeks złożony na kolumnach `x`, `z`	
11	Indeks złożony na kolumnach `y`, `z`	
12	Indeks złożony na kolumnach `x`, `y`, `z`	

2. Utworzyć tabelę:

```
DROP TABLE IF EXISTS indeksy2;

CREATE TABLE indeksy2 (
    id INT PRIMARY KEY AUTO_INCREMENT,
    imie VARCHAR(10),
    nazwisko VARCHAR(15),
    plec CHAR(1),
    zarobki INT,
    data_zatr DATE
);
```

Przygotować skrypt `indeksy2.sql` (np. przy pomocy języka programowania Python), który składać się będzie z instrukcji wstawiających do tabeli `indeksy2` 100000 rekordów z wartościami takimi, aby:

- kolumna `imie` została zapełniona losowo wybranymi imionami z listy 5 imion żeńskich i 5 imion męskich;
- kolumna `nazwisko` zawierała losowe frazy o długości pomiędzy 3 a 15; pierwsza litera jest duża;
- kolumna `plec` została zapełniona losowo literą `M` lub `K`;
- kolumna `zarobki` zawierała losową liczbę z przedziału od 1 do 5000;
- kolumna `data_zatr` zawierała losową datę pomiędzy datą bieżącą (dzień, w którym generowano skrypt), a datą maksymalnie 1000 dni wcześniejszą.

oraz przeprowadzić stosowne testy (analogicznie jak w zadaniu 1) pokazujące działanie różnych indeksów. Jakie zapytanie pokaże efektywność zdefiniowanych indeksów? Zapisać w tabeli, jaki indeks został utworzony i jaki jest czas wykonania zapytania. Należy pamiętać, aby wykonać testy dla różnych indeksów na tym samym zapytaniu.

3. Zdefiniować funkcję `F1`, która znajdzie i zwróci, ile jest książek, które nigdy nie zostały sprzedane (zamówienie nie zostało jeszcze zrealizowane lub nigdy ich nikt nie zamówił).
4. Zdefiniować funkcję `F2`, która znajdzie i zwróci, ile książek zostało zamówionych w roku podanym jako parametr funkcji.
5. Zdefiniować funkcję `F3`, która znajdzie i zwróci najstarszą datę zamówienia książek przez klienta o identyfikatorze podanym jako argument funkcji.
6. Zdefiniować funkcję `F4`, która obliczy i zwróci sumę iloczynów cen i ilości książek zamówionych, ale jeszcze nie zrealizowanych.
7. Zdefiniować funkcję `F5`, która dla identyfikatora dostawcy podanego jako argument sprawdzi i zwróci, ile różnych książek dostarczyli do księgarni.
8. Zdefiniować funkcję `F6`, która znajdzie i zwróci najstarszą alfabetycznie pierwszą literę imienia tych autorów, którzy w nazwisku na trzecim miejscu od końca mają literę przekazaną jako argument funkcji.
9. Zdefiniować funkcję `F7`, która znajdzie i zwróci najmniejszy identyfikator zrealizowanego zamówienia o największej wartości.

Rozdział 9

Transakcje

Transakcje to jedno z podstawowych pojęć współczesnych systemów (relacyjnych) baz danych. Umożliwiają one współbieżny dostęp do zawartości bazy danych, dostarczając niezbędnych mechanizmów synchronizacji. W tym rozdziale omówimy kontrolę współbieżności oraz teorię związaną z transakcjami. Nie jest naszym celem przywoływanie całej dostępnej na ten tema wiedzy, tylko wprowadzenie niezbędnych informacji, aby czytelnik mógł samodzielnie wykonać transakcję.

9.1. Kontrola współbieżności

Kontrola współbieżności ma miejsce w sytuacji, w której więcej niż jedno zapytanie musi jednocześnie zmienić dane. Aby zilustrować problem kontroli współbieżności, na potrzeby tego rozdziału rozważmy przykład skrzynki poczty elektronicznej stosowany w systemie Unix, w którym w pliku `mbox` wszystkie wiadomości są ze sobą połączone, tzn. znajdują się jedna za drugą. Z tego też względu odczyt i przetwarzanie wiadomości jest prosty, a nowa wiadomość dołączana jest na końcu pliku. Pojawia się pytanie: co się stanie, gdy dwa procesy spróbują jednocześnie dołączyć wiadomości do tej samej skrzynki pocztowej? W najgorszej sytuacji, może to doprowadzić do uszkodzenia skrzynki pocztowej i pozostawienia na końcu jej pliku dwóch pomieszanych ze sobą wiadomości. W celu uniknięcia tego rodzaju uszkodzeń, dobrze zaprojektowane systemy doręczania poczty elektronicznej używają blokad. Jeżeli klient próbuje dostarczyć wiadomość w chwili, gdy skrzynka pocztowa jest zablokowana, musi z doręczeniem tej wiadomości zaczekać aż do chwili zwolnienia blokady. Schemat blokad nie do końca zapewnia obsługę współbieżności, ponieważ w danej chwili tylko jeden proces może modyfikować skrzynkę pocztową, co w przypadku olbrzymich skrzynek pocztowych staje się problematyczne.

9.1.1. Blokady odczytu/zapisu

Odczyt wiadomości ze skrzynki pocztowej nie jest aż tak problematyczny, nawet gdy wielu klientów robi to jednocześnie. Problem pojawia się natomiast w momencie, gdy jeden z klientów chce usunąć na przykład piątą wiadomość, podczas gdy inny program w tym samym czasie odczytuje tę wiadomość - program ten może wówczas otrzymać uszkodzony bądź niespójny widok skrzynki pocztowej. Dlatego w celu zapewnienia bezpieczeństwa nawet odczyt skrzynki pocztowej wymaga szczególnej ostrożności.

Analogiczna sytuacja może mieć miejsce w bazie danych w prostej tabeli, gdy jednocześnie kliku klientów odczytuje, modyfikuje czy usuwa jej rekordy. System obsługujący jednoczesne operacje odczytu i zapisu zwykle posiada zaimplementowany mechanizm blokowania składający się z dwóch rodzajów blokad nazywanych blokadami współdzielonymi - blokadami wyłączonymi, bądź blokadami odczytu - blokadami zapisu. Blokady odczytu zakładane na zasobie są współdzielone, czyli wzajemnie się nie blokują - w tym samym czasie wielu klientów może odczytywać ten sam zasób, nie przeszkadzając sobie nawzajem. Z drugiej strony, blokady zapisu są wyłączne - tzn. blokują zarówno blokadę odczytu, jak i inne blokady zapisu. W danej chwili tylko jeden klient dokonuje zapisu w określonym zasobie, jednocześnie nie jest możliwa operacja odczytu tego zasobu w tym samym czasie. W bazach danych blokowanie występuje niemal nieustannie, a zarządzanie nimi jest przeprowadzane wewnętrznie w sposób, który praktycznie jest niezauważalny („przezroczysty"). Gdy klient przeprowadza modyfikację określonych danych na te dane nakładana jest blokada zapisu, co uniemożliwia innym klientom ich odczyt.

9.1.2. Zasięg działania blokad

Usprawnienie współbieżności można uzyskać poprzez selekcję blokowanych elementów. Zamiast zablokować cały zasób można zablokować jedynie tę jego część, która zawiera modyfikowane dane. Zminimalizowanie ilości danych blokowanych w danej chwili pozwala na to, aby zmiany określonego zasobu następowały jednocześnie, pod warunkiem, że ze sobą nie kolidują. Zwracamy uwagę na to, że blokady powodują zużywanie zasobów, gdyż należy pobrać blokadę, sprawdzić czy jest ona wolna, zwolnić ją itd. Zatem dobra strategia blokowania jest kompromisem między obciążeniem systemu, a bezpieczeństwem danych.

Większość dostępnych na rynku (komercyjnych) serwerów baz danych dostarcza mechanizmu blokady (ang. *lock*) na poziomie rekordu wraz z różnymi, często skomplikowanymi sposobami zachowania wydajności w przypadku zastosowania wielu blokad. Ważne jest odpowiednie ustalenie zakresu blokad na określonym poziomie, by uzyskać lepszą wydajność działania dla danych operacji, przy jednoczesnym pozostawieniu silnika nieco mniej dopasowanym do pozostałych celów. Dwie najważniejsze strategie blokowania w bazach danych to: blokada tabeli i blokada rekordu.

Blokada tabeli jest podstawową strategią blokowania, dostępną w bazie danych stanowiącą najmniejsze jej obciążenie. W przypadku blokowania całej tabeli klient może chcieć zapisać dane w tabeli (wstawić, usunąć, uaktualnić itp.) i wtedy wymagana jest blokada zapisu, która uniemożliwia wykonanie innych operacji odczytu i zapisu na modyfikowanych danych. W momencie gdy nikt nie przeprowadza operacji zapisu, możliwe jest nałożenie blokady odczytu, która nie koliduje z innymi blokadami odczytu. Blokada tabeli oferuje warianty powodujące zachowanie dobrej wydajności w określonych sytuacjach, na przykład blokada tabeli `READ LOCK` pozwala na wykonanie pewnego rodzaju operacji równoczesnego zapisu. Ponieważ blokada zapisu ma wyższy priorytet niż blokada odczytu, żądanie blokady zapisu zostanie umieszczone na początku kolejki blokad nawet wtedy, gdy w kolejce znajdują się już blokady odczytu. Blokada zapisu może wyprzedzić w kolejce blokady odczytu, natomiast blokady odczytu nie mogą wyprzedzić blokad zapisu. Przykładowo serwer wykorzystuje blokady na poziomie tabeli podczas wykonywania poleceń, takich jak `ALTER TABLE`, niezależnie od rodzaju silnika magazynu danych.

Z kolei blokada rekordu oferuje największy poziom współbieżności, przy czym stanowi jednocześnie największe obciążenie dla serwera. Jest ona implementowana na różne sposoby w silniku magazynu danych, a nie w serwerze. Zainteresowanego czytelnika rozszerzeniem wiadomości na temat blokad w PostgreSQL odsyłamy do dokumentacji systemu: `https://www.postgresql.org/docs/current/explicit-locking.html`.

9.2. Transakcje

Transakcja to grupa zapytań SQL wykonywanych niepodzielnie jako pojedyncza jednostka. Jeżeli silnik bazy danych może wykonać całą grupę zapytań w ramach transakcji, wówczas to zrobi. Jeśli natomiast choć jedno z zapytań nie będzie mogło zostać wykonane z powodu awarii, bądź jakiegokolwiek innego zdarzenia, wtedy nie będzie wykonane żadne z nich - zgodnie z zasadą „wszystko albo nic”. Prawidłowo zachowujący się system przetwarzania transakcji zachowuje cztery własności `ACID` (ang. ***A**tomic*, ***C**onsistency*, ***I**solation*, ***D**urability*), czyli jest niepodzielny, spójny, izolowany i trwały.

Jako przykład rozważmy aplikację bankową, w której wykonywanie określonych operacji wymusza konieczność istnienia transakcji. Zakładamy, że w bazie danych mamy dwie tabele (dwa konta) `Konto_X` i `Konto_Y`. W celu przelania `200 zł` z konta `Konto_X` na konto `Konto_Y` musimy wykonać następujące kroki:

1. Najpierw upewniamy się, że na koncie `Konto_X` znajdują się wystarczające środki (min. `200 zł`).
2. Następnie odejmujemy `200 zł` od salda konta `Konto_X`.
3. Aby dodać `200 zł` do salda konta `Konto_Y`.

Istnieje duże prawdopodobieństwo, że podczas wykonywania wyżej wymienionych instrukcji dojdzie do awarii i wówczas tylko część z nich zostanie ukończona. A co z pozostałymi? A co się stanie z naszymi pieniędzmi? Widzimy zatem, że cała operacja powinna być wykonywana jako transakcja, aby w przypadku niepowodzenia któregokolwiek kroku wszystkie dotychczas ukończone instrukcje zostały wycofane.

Ogólnie transakcja składa się z:

- Rozpoczęcia za pomocą polecenia:

  ```
  START TRANSACTION | BEGIN [TRANSACTION]
  ```

  Gdy użytkownik przyłącza się do bazy danych, rozpoczyna nową transakcję. Od tego momentu wszystkie operacje przez niego wykonywane są operacjami w ramach transakcji. Użytkownik może jawnie rozpocząć nową transakcję, żądając zakończenia transakcji bieżącej.
- Wykonania instrukcji.
- Zatwierdzenia poleceniem `COMMIT`. W momencie zatwierdzenia transakcji następuje zwolnienie blokad założonych przez operacje w ramach transakcji, usunięcie punktów bezpieczeństwa, sprawdzenie odroczonych ograniczeń integralnościowych. Zmiany, wprowadzone przez operacje w ramach transakcji zostają trwale zapisane w bazie danych i są widoczne dla innych transakcji. Po zatwierdzeniu bieżącej transakcji rozpoczęta zostaje nowa transakcja.
- Wycofania poleceniem `ROLLBACK`. W momencie wycofania bieżącej transakcji zostają anulowane wszystkie zmiany, wprowadzone do bazy danych przez operacje w ramach transakcji i następuje zwolnienie założonych blokad. Po wycofaniu bieżącej transakcji rozpoczynana jest nowa transakcja.
- Ustawienia opcjonalnych punktów zapisu za pomocą polecenia:

  ```
  SAVEPOINT <etykieta>
  ```

  dzięki którym możemy wycofać transakcję do pewnego punktu:

  ```
  ROLLBACK TO [SAVEPOINT] <etykieta>
  ```

  a nie do początku; efektem polecenia jest wycofanie zmian wprowadzonych przez operacje aktywnej transakcji od momentu utworzenia punktu bezpieczeństwa o podanej etykiecie do momentu wykonania polecenia wycofania. Jeśli wycofujemy się do punktu bezpieczeństwa wcześniejszego niż ostatnio utworzony, wszystkie punkty bezpieczeństwa utworzone później zostają usunięte; punkt bezpieczeństwa identyfikowany etykietą można usunąć jawnie z historii transakcji poleceniem:

  ```
  RELEASE SAVEPOINT <etykieta>
  ```

Zwracamy uwagę na to, że w momencie wycofania do określonego punktu zapisu, musimy kontynuować wykonywanie kolejnych instrukcji w transakcji jakbyśmy wpisywali i wykonywali je po raz pierwszy. Zaletą używania punktów zapisu jest możliwość cofnięcia się do określonego punktu w transakcji, gdy wykryjemy błąd, a nie anulowanie (wycofanie do początku) całej transakcji.
Zapiszemy teraz w postaci transakcji opisane wyżej trzy kroki przelewu kwoty 200 zł z jednego konta na drugie.

```
START TRANSACTION;
SELECT saldo FROM Konto_X
WHERE id_klienta = 12345;
UPDATE Konto_X SET saldo = saldo - 200.0
WHERE id_klienta = 12345;
UPDATE Konto_Y SET saldo = saldo + 200.0
WHERE id_klienta = 12345;
COMMIT;
```

9.2.1. ACID

Transakcja musi funkcjonować jako pojedyncza, niepodzielna jednostka, tak aby cała transakcja została albo zatwierdzona, albo odrzucona. Kiedy transakcja jest niepodzielna, wówczas nie istnieje coś takiego jak częściowo zakończona transakcja. Obowiązuje zasada „wszystko albo nic", a o transakcji mówi się, że jest „atomowa".

Wykonanie transakcji powinno przeprowadzać bazę danych z jednego stanu spójnego do kolejnego. Spójność gwarantuje, że ewentualna awaria pomiędzy instrukcjami 3 i 4 z przykładu ze strony 118 nie spowoduje zniknięcia pieniędzy z konta `Konto_X` klienta. W przypadku awarii transakcja nie zostanie zatwierdzona, a żadna zmiana wykonana przez transakcję nie będzie naniesiona w bazie danych. Aż do chwili zakończenia transakcji jej wynik jest zwykle niewidoczny dla innych transakcji. Takie rozwiązanie sprawia, że jeżeli między wykonaniem wierszy 3 i 4 z przykładu ze strony 118 nastąpi uruchomienie procedury sprawdzającej saldo konta, wynik będzie nadal pokazywał `200 zł` na koncie `Konto_X`. Użycie słowa „zwykle" stanie się zrozumiałe po omówieniu poziomów izolacji. Po zatwierdzeniu transakcji wprowadzone przez nią zmiany są trwałe. Oznacza to, że zmiany muszą być zapisane, aby dane nie zostały utracone w przypadku awarii systemu. Dzięki transakcjom zgodnym z `ACID` zwiększa się poziom bezpieczeństwa bazy danych. Wadą jest fakt wykonywania przez serwer bazy danych dodatkowych skomplikowanych operacji, z których istnienia użytkownik często nawet nie zdaje sobie sprawy. W związku z tym serwer bazy danych obsługujący transakcje ACID ogólnie wymaga większej mocy procesora, pamięci oraz miejsca na dysku twardym niż serwer nie obsługujący

tego typu transakcji. W przypadku baz danych to użytkownik decyduje, czy aplikacja wymaga transakcji, czy może innego rodzaju poziomu bezpieczeństwa.

9.2.2. Poziomy izolacji

Standard SQL definiuje cztery poziomy izolacji wraz z określonymi regułami opisującymi zmiany, które są lub nie są widoczne wewnątrz i na zewnątrz transakcji. Niższy poziom izolacji zwykle pozwala na wyższy poziom współbieżności i jednocześnie stanowi mniejsze obciążenie dla systemu. Każdy silnik magazynu danych nieco inaczej implementuje poziomy izolacji, należy więc zapoznać się z podręcznikiem użytkownika silnika, który ma być używany. Na potrzeby tego rozdziału zostaną krótko i ogólnie przedstawione informacje o czterech poziomach izolacji.

READ UNCOMMITTED

Na poziomie izolacji `read uncommitted` transakcje mają wgląd do wyników niezatwierdzonych jeszcze transakcji. W tym przypadku istnieje możliwość wystąpienia anomalii zwanej „brudnym odczytem" (ang. *dirty read*), gdy transakcja odczytuje dane, zmienione przez inną transakcję, która potem zostaje wycofana. Przez to pierwsza transakcja odczytuje dane, które już nie istnieją. Poziom ten jest bardzo rzadko stosowany w praktyce, ponieważ wydajność nie jest znacząco większa od osiąganej na innych poziomach i jednocześnie wymaga od programisty wiedzy i pełnej świadomości tego, co robi.

READ COMMITTED

Domyślnym poziomem izolacji dla większości systemów (relacyjnych) baz danych (w tym dla PostgreSQL) jest poziom `read committed`. Transakcja na tym poziomie izolacji może odczytywać tylko te zmiany wprowadzone przez inne transakcje, które zostały już zatwierdzone. Natomiast zmiany wykonywanej transakcji nie będą widoczne aż do chwili jej zatwierdzenia. Ten poziom izolacji pozwala na „odczyt niepowtarzalny" (ang. *non-repeatable read*) - to samo polecenie wydane dwukrotnie może za każdym razem zwrócić inne dane, gdyż równolegle inna transakcja modyfikuje wartości atrybutów tego samego rekordu.

REPEATABLE READ

Poziom `repeatable read` rozwiązuje problem odczytu „brudnych danych" występujący na poziomie `read uncommitted`. Gwarantuje, że dowolny rekord odczytywany w transakcji będzie „wyglądał tak samo" podczas kolejnych jego odczytów w tej samej transakcji. Istnieje jednak niebezpieczeństwo odczytu tzw. „fantomów" (ang. *phantom read*). Odczyt fantomów może wystąpić, kiedy po zaznaczeniu pewnego zakresu rekordów inna transakcja wstawi nowy rekord w zaznaczonym zakresie, a później nastąpi ponowne zaznaczenie tego samego zakresu. W takim przypadku użytkownik będzie widział nowy, wstawiony przed chwilą rekord, czyli fantom.

SERIALIZABLE

Najwyższym poziomem izolacji jest `serializable`, który rozwiązuje problem odczytu fantomów poprzez wymuszenie wykonywania transakcji po kolei, a więc nie mogą one ze sobą kolidować. Symuluje zatem szeregowe wykonanie transakcji (jedna po drugiej). Poziom ten nakłada blokadę na każdy odczytywany rekord i przez to może wystąpić duża liczba sytuacji przekroczenia czasu oczekiwania lub konflikt blokad.

Aby zmienić poziom izolacji dla bieżącej transakcji, należy wykonać polecenie:

```
SET TRANSACTION ISOLATION LEVEL <poziom>;
```

W tabeli 9.1 zbiorczo przedstawiamy poziomy izolacji wraz z anomaliami, które mogą w nich wystąpić.

Tabela 9.1: Poziomy izolacji i ich wady

Poziom izolacji	**Brudny odczyt?**	**Odczyt niepowtarzalny?**	**Odczyt fantomów?**	**Blokady odczytu?**
READ UNCOMMITTED	Tak	Tak	Tak	Nie
READ COMMITED	Nie	Tak	Tak	Nie
REPEATABLE READ	Nie	Nie	Tak	Nie
SERIALIZABLE	Nie	Nie	Nie	Tak

9.2.3. Zakleszczenie

Poważnym problemem, występującym w bazach danych ze zbiorem współbieżnych transakcji, jest możliwość wystąpienia „zakleszczenia" (ang. *deadlocks*). Zakleszczenie („wzajemna blokada" lub „śmiertelny uścisk") - występuje wtedy, gdy dwie lub większa liczba transakcji wzajemnie się wstrzymują i żądają nałożenia blokady na ten sam zasób, w wyniku czego powstaje cykl zależności. Zakleszczenia występują, jeśli transakcje próbują nałożyć blokady na zasób w różnej kolejności. Mogą także wystąpić, gdy większa liczba

transakcji nakłada blokadę na ten sam zasób. Powstaje zatem sytuacja, w której co najmniej dwie transakcje czekają wzajemnie na zwolnienie zasobów, przez co żadna z nich nie może zakończyć swojego działania.

Tabela 9.2: Przykład zakleszczenia transakcji

Użytkownik: 12345 Transakcja #1	Użytkownik: 1234 Transakcja #2

START TRANSACTION;
  UPDATE Konto_X
  SET saldo = saldo - 200
  WHERE id_klienta = 12345;
  UPDATE Konto_Y
  SET saldo = saldo + 200
  WHERE id_klienta = 1234;
COMMIT;
	START TRANSACTION;
	  UPDATE Konto_Y
	  SET saldo = saldo - 200
	  WHERE id_klienta = 1234;
	  UPDATE Konto_X
	  SET saldo = saldo + 200
	  WHERE id_klienta = 12345;
	COMMIT;

Zilustrujemy zakleszczenie na przykładzie przedstawionym w tabeli 9.2. Zakładamy, że dwaj właściciele kont bankowych o identyfikatorach `12345` i `1234` wykonują wzajemny przelew pieniędzy. W tym celu każdy z nich uruchamia na swoim komputerze klienta `psql` (aby to zasymulować czytelnik powinien uruchomić klienta `psql` w dwóch odrębnych oknach (terminalach)). Użytkownik `12345` rozpoczął swoją transakcję (`Transakcja #1`), w której wykonuje pierwsze polecenie modyfikujące stan swojego konta (`Konto_X`), zmniejszając go o kwotę przelewu. Przed modyfikacją rekord zostaje zablokowany w trybie wyłącznym przez `Transakcja #1`. W międzyczasie użytkownik `1234` również rozpoczął swoją transakcję (`Transakcja #2`) i w pierwszym poleceniu zmodyfikował stan swojego konta (`Konto_Y`), zmniejszając go o kwotę przelewu. Tu również rekord zostaje zablokowany przez `Transakcja #2`. Użytkownik `12345` w swojej `Transakcja #1` kontynuuje pracę i próbuje zmodyfikować stan konta (`Konto_Y`) użytkownika `1234`, zwiększając go o kwotę przelewu. Jednak transakcja nie może uzyskać blokady, bo rekord ten został wcześniej zablokowany przez `Transakcja #2`. `Transakcja #1` zostaje zawieszona, czekając na zwolnienie blokady. Tymczasem użytkownik `1234` w `Transakcja #2` chce zwiększyć konto (`Konto_X`) użytkownika `12345` o kwotę przelewu. Operacja jednak nie może zostać zrealizowana, gdyż transakcja nie uzyskuje blokady na rekordzie, który ma być zmodyfikowany – rekord ten został wcześniej zablokowany przez `Transakcja #1`. `Transakcja #2` zostaje również zawieszona i czeka na zakończenie `Transakcja #1`. Obie transakcje zostają zawieszone, czekając na zdjęcie blokad. Żadna nie może kontynuować pracy, gdyż nie może

uzyskać blokad. Przez to nie może również zwolnić blokad, na które czeka druga transakcja. Wystąpiło zakleszczenie. System zarządzania bazą danych wykrywa zakleszczenie i rozwiązuje je, wycofując jedno z czekających poleceń zawieszonych transakcji. Dalej to użytkownik musi zdecydować, co ma się dziać z transakcjami: czy je wycofać, czy może kontynuować,wykonując inne polecenia.

Istnienie zakleszczenia potwierdza najczęściej wyjątkowo wolne przetwarzanie zapytań. Systemy baz danych implementują różnego rodzaju formy wykrywania zakleszczeń i upłynięcia czasu danej operacji. Bardziej zaawansowane systemy baz danych wykrywają wzajemne zależności i natychmiast zwracają błąd. Inne systemy powodują przerwanie operacji po upływie określonego okresu czasu lub mogą wycofać transakcję, która ma mniejszą liczbę rekordów zablokowanych na wyłączność (którą w związku z tym łatwiej wycofać).

Zachowanie w trakcie blokowania oraz jego kolejność są charakterystyczne dla danego silnika magazynu danych, tak więc niektóre silniki mogą zakleszczyć się podczas wykonywania określonej sekwencji poleceń, a inne w takiej sytuacji nie ulegną zakleszczeniu. Niektóre zakleszczenia są nie do uniknięcia z powodu faktycznych konfliktów danych, natomiast inne powstają w wyniku sposobu działania danego silnika magazynu danych. Zakleszczenie nie może być przerwane bez wycofania jednej z transakcji albo częściowo, albo całkowicie. Pamiętaj: zakleszczenia są rzeczywistością w systemach transakcyjnych i aplikacja powinna być przygotowana na ich obsługę. Wiele z nich po prostu próbuje na nowo wykonać daną transakcję.

Dzięki rejestrowaniu zdarzeń transakcji, ich działanie jest znacznie bardziej efektywne. Zamiast uaktualniać tabele na dysku twardym po każdej wprowadzonej zmianie, silnik magazynu danych może wprowadzić zmiany w kopii danych umieszczonych w pamięci - taka operacja jest bardzo szybka. Następnie silnik magazynu danych może zapisać zmiany rekordu w dzienniku zdarzeń transakcji znajdującym się na dysku, a więc trwałym - to również jest relatywnie szybka operacja, ze względu na wykorzystanie ciągłych operacji I/O na małym obszarze dysku twardego zamiast wielu losowych operacji I/O rozrzuconych w różnych miejscach dysku. Co pewien czas następuje uaktualnienie tabeli zapisanej na dysku twardym.

Większość silników magazynu danych wykorzystujących technikę rejestrowania zapisu z wyprzedzeniem (ang. *write-ahead logging*), dwukrotnie zapisuje zmiany na dysku twardym. Jeżeli awaria nastąpi po zapisaniu zmian w dzienniku zdarzeń transakcji, ale przed wprowadzeniem tych zmian na rzeczywistych danych, silnik magazynu danych może odzyskać zmiany po ponownym uruchomieniu, stosując w tym celu różne, zależne od silnika, metody.

9.2.4. Transakcje w PostgreSQL - AUTOCOMMIT

PostgreSQL domyślnie działa w trybie AUTOCOMMIT - jeśli użytkownik wyraźnie nie zainicjował transakcji, każde zapytanie będzie automatycznie wykonywane jako oddzielna transakcja. Włączenie (on) lub wyłączenie (off) trybu AUTOCOMMIT dla bieżącego połączenia jest możliwe poprzez ustawienie wartości zmiennej:

```
postgres=# \set AUTOCOMMIT off;
postgres=# \echo :AUTOCOMMIT
postgres=# \set AUTOCOMMIT on;
```

Po ustawieniu zmiennej w postaci AUTOCOMMIT off użytkownik zawsze będzie w transakcji, aż do wydania polecenia COMMIT lub ROLLBACK. Następnie baza danych rozpocznie kolejną transakcję. Pełna lista poleceń powodujących automatyczne zatwierdzenie transakcji powinna znajdować się w dokumentacji dotyczącej odpowiedniej wersji PostgreSQL.

9.3. Zadania do samodzielnego rozwiązania

1. Dodać punkty zapisu po każdej instrukcji w transakcji ze strony 114 realizującej przelew bankowy. Przetestować działanie transakcji z cofaniem się do poszczególnych punktów zapisu, aby na koniec całkowicie wycofać transakcję do jej początku.
2. Wykonać jeszcze raz zadanie 1, ale jednocześnie uruchomić klienta psql w drugim oknie (terminalu) i zalogować się na swoje konto, aby wyświetlać w nowej rozpoczętej tam transakcji zawartość tabel Konto_X oraz Konto_Y po każdej instrukcji modyfikacji stanów kont w pierwszej transakcji. Zaobserwować działanie domyślnie ustawionego poziomu izolacji.
3. Wykonać samodzielnie przykład zakleszczenia z tabeli 9.2, tworząc dwie sesje tego samego użytkownika i sprawdzić, w jaki sposób PostgreSQL radzi sobie z tym problemem.
4. Rozpocząć nową transakcję. Zwiększyć cenę książki o identyfikatorze ISBN = 1 o 500 złotych. Utworzyć punkt bezpieczeństwa S1. Zwiększyć cenę książki o identyfikatorze ISBN = 15 o 300 złotych. Utworzyć punkt bezpieczeństwa S2. Usunąć książkę o identyfikatorze ISBN = 4. Wycofać transakcję do punktu S1 i zobaczyć zawartość tabeli t_ksiazka. Spróbować wycofać transakcję do punktu S2. Na koniec wycofać całą transakcję.
5. Rozpocząć nową transakcję. Usunąć autora o nazwisku Mickiewicz, utworzyć punkt bezpieczeństwa S1, a następnie zmienić typ atrybutu tytul w relacji t_ksiazka na VARCHAR(60). Spróbować wycofać transakcję do punktu S1. Opisać zaobserwowane zmiany.

6. Uruchomić dwie sesje tego samego użytkownika, a w nich rozpocząć transakcje. W pierwszej sesji wykonać polecenie, zwiększające cenę książki o `ISBN = 2` o `10%`, w drugiej sesji dla tej samej książki zwiększyć jej cenę o `20%`. Co zaobserwowałeś w drugiej sesji? Zatwierdzić zmiany w pierwszej sesji i cenę książki o `ISBN = 2`. W drugiej sesji odczytać cenę książki o `ISBN = 2`, a następnie zatwierdzić zmiany. Odczytać ponownie cenę książki o `ISBN = 2` w pierwszej sesji.
7. Uruchomić dwie sesje tego samego użytkownika, a w nich rozpocząć transakcje. W pierwszej sesji określić poziom izolacji transakcji na `serializable`:
   ```
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   ```
   i odczytać informacje o 5 początkowych książkach ułożonych w porządku rosnącym według `ISBN`. W drugiej sesji zwiększyć dwukrotnie cenę książki o `ISBN = 1`. Zatwierdzić zmianę. Sprawdzić ponownie, jaki obraz relacji `t_ksiazka` widzi pierwsza z uruchomionych sesji. Spróbować w tej sesji zwiększyć cenę książki o `ISBN = 1` o `50%`. Opisać zaobserwowane zmiany.

Rozdział 10

Tworzenie aplikacji bazodanowych w języku Java

Od samego początku twórcy języka Java dążyli do stworzenia mechanizmów, które umożliwiłyby programistom tworzenia aplikacji Javy korzystających z baz danych. W efekcie firma Sun Mikrosystems opracowała interfejs JDBC (ang. *Java Database Connectivity*) – interfejs programowania aplikacji (ang. API), który do tego celu służy.

W tym rozdziale przedstawimy te elementy języka Java, które niezbędne są do powstania aplikacji opartej o bazę danych.

10.1. Przygotowanie środowiska pracy

Przy tworzeniu aplikacji bazodanowych w języku Java korzysta się z trzech komponentów:

- Systemu zarządzania relacyjnymi bazami danych – w naszym przypadku PostgreSQL. Instalacja systemu zarządzania bazami danych została już opisana w podrozdziale 1.2.
- JDK (ang. *Java Development Kit*) – zestaw narzędzi i bibliotek potrzebnych do tworzenia, debugowania i uruchamiania aplikacji w języku Java.
- Środowiska programistycznego – środowisko programistyczne (IDE) ułatwiające tworzenie i późniejszy rozwój aplikacji. Do prezentacji przykładów zostaną wykorzystane dwa środowiska: Intellij Idea oraz Visual Studio Code (VSC).

W przypadku wyboru środowiska Visual Studio Code warto zainstalować dwa kolejne narzędzia:

- Rozproszony system kontroli wersji Git – przydatny program do śledzenia zmian w kodzie źródłowym podczas tworzenia oprogramowania. Git umożliwia zespołom deweloperów pracę nad wspólnymi projektami, w tym: śledzenie historii zmian, roz-

wiązywanie konfliktów, zarządzanie wersjami i umożliwienie pracy na wielu gałęziach projektu równocześnie. Przechowywanie projektów w repozytoriach stanowi dodatkowe zabezpieczenie w razie uszkodzenia plików projektu.

- Apache Maven – narzędzie do zarządzania i automatyzacji budowania projektów, głównie w języku Java. Maven oferuje szereg funkcji: kompilację, testowanie, wdrażanie.

Do tworzenia aplikacji bazodanowej wykorzystamy JDK w wersji 22 (najnowszej, dostępnej w trakcie pisania tego skryptu). Pliki instalacyjne są dostępne na stronie: `https://www.oracle.com/java/technologies/downloads/#java22`. Wystarczy wybrać platformę systemową i preferowany plik instalacyjny, np. dla systemu Windows x64 MSI Installer. Java JDK zawiera wszystkie niezbędne narzędzia do tworzenia (kompilacji), debagowania, uruchamiania aplikacji, włączenie ze środowiskiem uruchomieniowym Java Runtime Environment (JRE). Środowisko uruchomieniowe zawiera JVM (ang. *Java Virtual Machine*), klasy biblioteczne oraz inne pliki pomocnicze potrzebne do uruchamiania aplikacji Java.

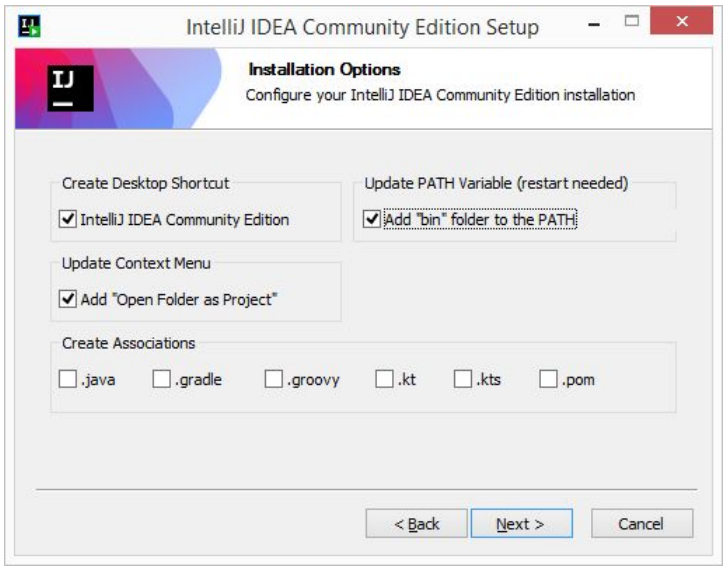
Instalacja JDK w systemie Windows odbywa się z użyciem prostego kreatora, w którym jedynie można wskazać miejsce instalacji na dysku. Domyślnie jest to folder „Program files" na dysku C. Jeżeli nie ma ważnych powodów, to nie należy go zmieniać. Warto JDK zainstalować przed instalacją środowiska programistycznego, ponieważ instalator środowiska odnajdzie w systemie położenie plików JDK i wykona konfigurację w sposób automatyczny.

Środowisko programistyczne IntelliJ IDEA można pobrać ze strony: `https://www.jetbrains.com/idea/download/?section=windows`. Będziemy korzystać z bezpłatnej wersji IntelliJ IDEA Community Edition. Instalacja programu w systemie Windows odbywa się przy pomocy kreatora, w którym można określić miejsce instalacji na dysku oraz opcje instalacji: utworzenie skrótu, dodanie do zmiennej środowiskowej PATH ścieżki do podkatalogu bin, dodanie do menu podręcznego możliwości otwarcia folderu jako projektu w programie IntelliJ, czy też powiązanie wybranych rozszerzeń plików z programem (rys. 10.1).

Plik instalacyjny aplikacji Visual Studio Code jest dostępny na stronie: `https://code.visualstudio.com/download`. Należy wybrać wersję odpowiednią dla swojej platformy systemowej i sprzętowej. Dla systemu Windows i komputerów PC najlepszym wyborem będzie System installer x68. Instalacja odbywa się przy użyciu kreatora i sprowadza się praktycznie do zatwierdzania kolejnych kroków instalacji.

Aplikacja Apache Maven dostępna jest pod adresem `https://maven.apache.org/download.cgi` w postaci pliku archiwum (wersje binarne i źródłowe). Dla wersji binarnej należy rozpakować archiwum i skopiować katalog programu do podkatalogu „C:\Program files". Dobrze jest dodać podkatalog bin Mavena do zmiennej środowiskowej `PATH`.

Wersja instalacyjna programu Git dla systemu Windows znajduje się na stronie `https://www.git-scm.com/download/win`. Do wyboru mamy wersję instalowaną lub przenośną (ang. *portable*), zarówno dla komputerów 32-, jak i 64-bitowych. Pobieramy plik 64-bit Git for Windows Setup i uruchamiamy kreator. Kreator składa się z kilku kroków i daje możliwości wyboru wielu opcji. My wybieramy ustawienia domyślne, które są wystarczające na nasze potrzeby.

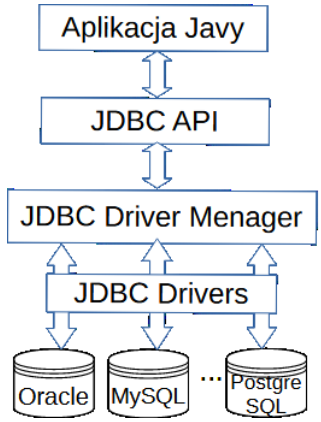


Rysunek 10.1: Okno wyboru opcji przy instalacji środowiska IntelliJ Idea

10.2. Java JDBC

Twórcy Javy zdecydowali się na użycie podobnego do sprawdzonego rozwiązania dostępu do baz danych ODBC (ang. *Open Database Connectivity*) opracowanego przez firmę Microsoft. Idea polega na tym, że programy Javy komunikują się z programem zarządzającym w celu skorzystania z uprzednio zarejestrowanego sterownika dostępu do bazy danych. Dzięki temu można tworzyć aplikacje bazodanowe w języku Java, które mogą używać baz danych różnych producentów. Wystarczy, że producent udostępni sterownik do swojej bazy danych. Schemat architektury JDBC zamieszczono na rysunku 10.2.

1. Aplikacja Javy przy pomocy JDBC API komunikuje się z bazą danych, korzysta zarówno z menedżera sterowników, jak i z samych sterowników.
2. JDBC API dostarcza niezbędnych klas i interfejsów umożliwiających komunikację aplikacji Java z bazą danych.



Rysunek 10.2: Architektura JDBC

3. Menadżer sterowników JDBC umożliwia zarządzanie sterownikami różnych firm. Dzięki temu aplikacja Javy w trakcie swojego działania może używać wielu baz danych różnych producentów.
4. Sterownik jest pośrednikiem pomiędzy aplikacją a bazą danych. Tłumaczy polecenia aplikacji Javy na polecenia zrozumiałe dla systemu bazy danych. Również dokonuje konwersji odwrotnej – przygotowuje wyniki zwrócone przez bazę do postaci „zrozumiałej" dla aplikacji Javy.

10.3. Rodzaje sterowników w JDBC

Specyfikacja JDBC określa cztery typy sterowników:

- Typ 1 tłumaczy JDBC na ODBC i używa sterowników ODBC do komunikacji z bazą danych. Ten typ nie jest polecany.
- Typ 2 napisany jest częściowo w języku Java, a częściowo w kodzie macierzystym danej platformy. Komunikuje się z bazą danych, korzystając z interfejsu programowego klienta danego systemu baz danych. Konieczne jest zainstalowanie również fragmentów oprogramowania specyficznego dla danej platformy.
- Typ 3 napisany jest w języku Java i korzysta z protokołu komunikacji niezależnego od systemu bazy danych. Komunikacja ta odbywa się z komponentem serwera, który tłumaczy żądania sterownika na specyficzny protokół określonego systemu bazy danych. Oprogramowanie po stronie klienta jest niezależne od systemu bazy danych.
- Typ 4 napisany jest w języku Java i tłumaczy żądania JDBC na specyficzny protokół danego systemu bazy danych.

Większość producentów systemów baz danych udostępnia sterowniki typu 3 lub 4.

10.4. Szkielet aplikacji bazodanowej w Javie

Aby aplikacja Javy mogła używać bazy danych należy wykonać kilka kroków:

1. Ustanowić połączenie z bazą danych.
2. Utworzyć zapytanie.
3. Wykonać zapytanie.
4. Przetworzyć wyniki zapytania.
5. Zamknąć połączenie z bazą danych.

W trakcie działania aplikacji mogą być tworzone różne zapytania i w różny sposób przetwarzane wyniki. Kroki 2-4 mogą być wykonywane wielokrotnie.

10.4.1. Przykład – nawiązywanie połączenia z bazą danych

Obecnie do tworzenia aplikacji używa się różnych środowisk programistycznych, które ułatwiają pracę programistów. W tym podrozdziale pokażemy, jak utworzyć projekt w dwóch przykładowych, popularnych i dostępnych na licencji wolnego oprogramowania środowiskach programistycznych: IntelliJ Idea oraz Visual Studio Code. Dodatkowo aplikacje te dostępne są na najpopularniejsze systemy operacyjne: Windows, Linux oraz macOS.

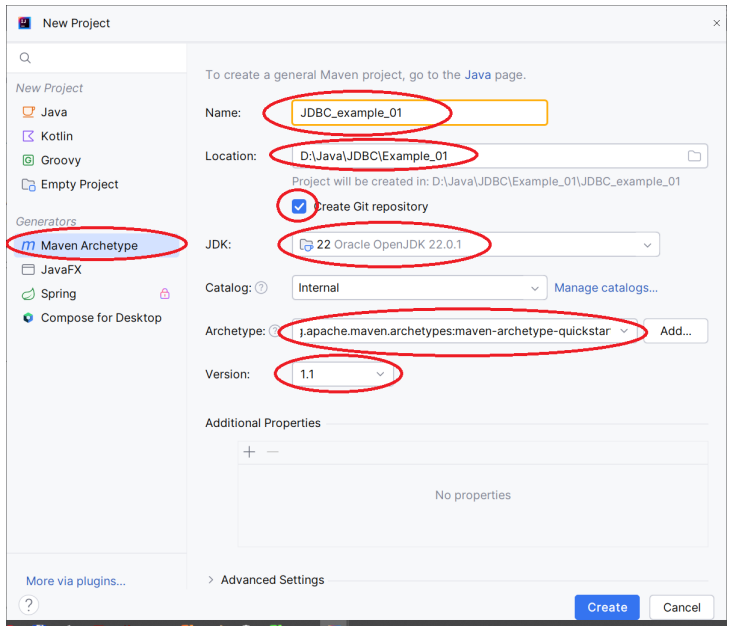
Projekt w programie INtelliJ Idea

Zaczynamy od utworzenia nowego projektu w programie IntelliJ Idea. W oknie kreatora (rys. 10.3) wybieramy nazwę projektu, lokalizację plików projektu, zaznaczamy opcję tworzenia repozytorium Git, wybieramy zainstalowaną wersję JDK, Maven Archetype oraz jako archetyp (szablon projektu, który zawiera przedefiniowaną strukturę katalogów, pliki konfiguracyjne oraz zależności, które są niezbędne do budowy wybranego rodzaju projektu) org.apache.maven.archetypes:maven-archetype-quickstart w najnowszej wersji. Po utworzeniu projektu może pojawić się okienko z komunikatem Microsoft Defender Configuration, wówczas klikamy na Automatically w celu dodania naszej aplikacji do listy wyjątków programu Defender.

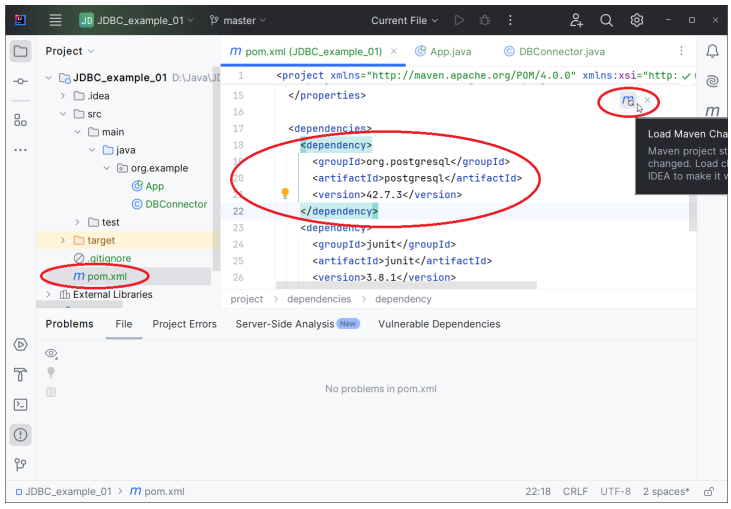
W następnej kolejności dodamy do projektu zależność (ang. *dependency*), która reprezentuje sterownik dla bazy danych PostgreSQL (rys. 10.4). W projekcie Mavena zależności dodajemy do pliku pom.xml (listing 10.1).

Listing 10.1: *Sekcja opisująca zależność postgresql*

```
<dependency>
     <groupId>org.postgresql</groupId>
     <artifactId>postgresql</artifactId>
     <version>42.7.3</version>
</dependency>
```



Rysunek 10.3: Okno kreatora nowego projektu w aplikacji IntelliJ Idea



Rysunek 10.4: Kod zależności dla sterownika bazy danych PostgreSQL

Następnie zmiany należy zatwierdzić, klikając na przycisk (Load Maven Changes – rysunek 10.4), bez tego nie będą obowiązywały. Nie powoduje to jeszcze załadowania sterownika bazy, do tego celu służy menadżer sterowników. Dostępne wersje sterownika możemy sprawdzić na stronie `https://jdbc.postgresql.org`.

Zdefiniujmy klasę, której zadaniem będzie nawiązywanie i kończenie połączenia z bazą danych (listing 10.2).

Do nawiązywania połączenia z bazą danych służy klasa `Connection`. Reprezentuje ona otwarte połączenie do źródła danych i jest używana do wykonywania instrukcji SQL oraz zarządzania transakcjami. Obiekt klasy `Connection` jest zwracany przez metodę `getConnection` klasy `DriverManager`, która zarządza sterownikami bazy danej. Połączenie następuje z bazą danych, której lokalizacja jest wskazana przez adres URL. Logowanie następuje dla wskazanego użytkownika bazy i hasła. Ważne jest, aby przed uruchomieniem programu Javy:

- uruchomić serwer bazy danych (PostgreSQL);
- stworzyć bazę danych o podanej nazwie (w typ przypadku `test_00`) i dostępnej na porcie numer 5432 na komputerze lokalnym;
- utworzyć użytkownika o podanym loginie (`postgres`) i haśle (`12345`).

LISTING 10.2: *Klasa odpowiedzialna za nawiązanie i zakończenie połączenia z bazą danych*

```
1  package org.example;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.SQLException;
6  public class DBConnector {
7    private static final String URL = "jdbc:postgresql://localhost
       :5432/test_00";
8    private static final String user = "postgres";
9    private static final String password = "12345";
10   public static Connection connect() throws SQLException {
11      Connection connection =
12         DriverManager.getConnection(URL, user, password);
13      System.out.println("Połączono");
14      return connection;
15   }
16   public static void close(Connection connection) throws
       SQLException {
17      connection.close();
18   }
19 }
```

Po nawiązaniu połączenia na ekranie pojawi się napis „Połączono", który służy jedynie celom demonstracyjnym i w zwykłym programie jest on zbędny. W przypadku braku połączenia zostanie wygenerowany wyjątek klasy `SQLException`. Do przetestowania połączenia z bazą danych napiszemy prosty program – listing 10.3. W metodzie `main` następuje wywołanie statycznej metody `connect` z klasy `DBConnector` w celu nawiązania połączenia z bazą danych. Kod programu odpowiedzialny za pracę z bazą danych został umieszczony w sekcji `try-catch-finally`, ponieważ w trakcie działania programu mogą pojawić się sytuacje wyjątkowe, które wymagają obsłużenia. Należą do nich między innymi: problem z dostępem do bazy danych (np. otwarcie lub zamknięcie) lub błędy związane z zapytaniami SQL, np. błąd składni, błąd transakcji.

W wierszu 15 został utworzony obiekt `stat` typu `Statement`, który umożliwia wykonanie poleceń SQL na podłączonej bazie danych. Za realizację poleceń SQL odpowiada metoda `executeUpdate` oraz `executeQuery` z tej klasy. W przykładowym programie wykonane są cztery polecenia SQL: jedno tworzy tabelę `Greetings` z polem `Message`, drugie umieszcza w tabeli napis „Witaj świecie!", trzecie polecenie – zapytanie, pobiera całą zawartość tabeli `Greetings` do obiektu typu `ResultSet`, a czwarte usuwa utworzoną tabelę z bazy danych.

Interfejs `ResultSet` umożliwia dostęp do tabeli, która zawiera wyniki zapytania. Wierszy w tabeli wyników może być wiele, dostęp do kolejnych wierszy jest poprzez metodę `next`, która przesuwa kursor do kolejnego wiersza lub zwraca wartość `null`, gdy kolejnego wiersza nie ma. Możemy odczytać poszczególne wartości tabeli przy pomocy metod dostępowych z interfejsu `ResultSet`, które obsługują wszystkie typy proste oraz łańcuchy znaków, np. `getInt`, `getBoolean`, `getString`. Wartości można pobrać, podając jako parametr metody dostępowej numer lub nazwę kolumny. Kolumny są numerowane od 1. Aby zapewnić maksymalną przenośność, kolumny zestawu wyników w każdym wierszu należy czytać w kolejności od lewej do prawej, a każdą kolumnę należy czytać tylko raz.

Za przetwarzanie danych odpowiedzialna jest pętla `while` w wierszu 21, gdzie metoda `next` jest odpowiedzialna za przeglądanie kolejnych wierszy tablicy wyników, a metoda `getString` odczytuje wartość pierwszej kolumny tablicy wyników.

Listing 10.3: *Klasa publiczna z kodem sprawdzającym połączenia z bazą danych*

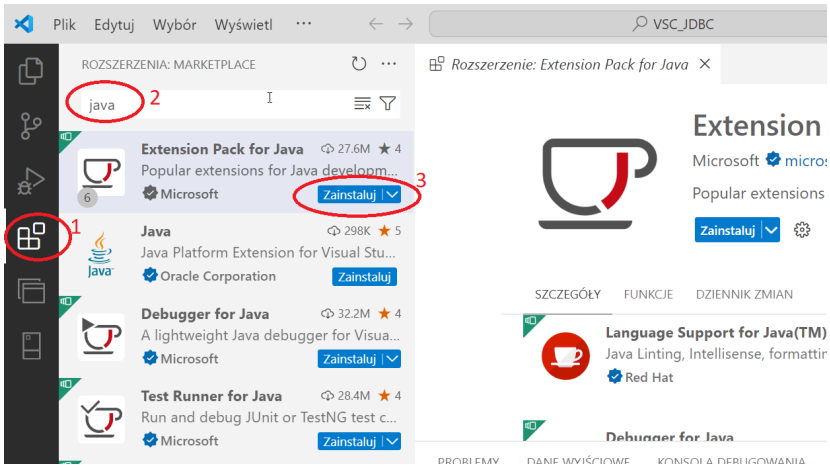
```
1 package org.example;
2
3 import java.sql.Connection;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.sql.Statement;
7
8 public class Example_01 {
```

```
9   public static void main( String[] args ) {
10    Connection connection = null;
11    ResultSet result = null;
12    try {
13      connection = DBConnector.connect();
14      Statement statement = connection.createStatement();
15      statement.executeUpdate("CREATE TABLE Greetings (Message
      CHAR(20))");
16      statement.executeUpdate("INSERT INTO Greetings VALUES ('
      Witaj, świecie!')");
17      result = statement.executeQuery("SELECT * FROM Greetings");
18      while (result.next())
19        System.out.println(result.getString(1));
20      statement.executeUpdate("DROP TABLE Greetings");
21    }
22    catch (SQLException e) {
23      System.err.println("Stan SQL: " + e.getSQLState());
24      System.err.println("Kod błędu: " + e.getErrorCode());
25      System.err.println(e.getMessage());
26      Throwable t = e.getCause();
27      while (t != null) {
28         System.out.println("Przyczyna wyjątku: " + t);
29         t = t.getCause();
30      }
31    }
32    finally {
33      try {
34         if (result != null) {
35           result.close();                  // zwolnienie zasobów
36         }
37         if (connection != null) {
38            DBConnector.close(connection); // zamknięcie połą
      czenia
39         }
40      }
41      catch (SQLException e) {
42        e.printStackTrace();
43      }
44    }
45  }
46 }
```

Projekt w Visual Studio Code

Na początku należy przygotować VSC do wygodnej pracy przy tworzeniu aplikacji Javy. Siłą środowiska VSC jest możliwość korzystania z rozszerzeń (ang. *extension*), które znacznie zwiększają możliwości środowiska. Tworząc aplikacje Javy, musimy zainstalować rozszerzenie Java Extension Pack, natomiast zalecane jest jeszcze zainstalowanie rozszerzenia Language Support for Java (TM) by Red Hat (między innymi uzupełnianie kodu, wsparcie dla Mavena i Grandla). Rozszerzenie Maven for Java będzie wymagane, gdy będziemy tworzyć projekty z użyciem aplikacji Maven.

Rozszerzenie możemy dodać, klikając na przycisku rozszerzeń (rys 10.5), następnie wpisując frazy związanej z rozszerzeniem, np. Java, a następnie wyszukanie na liście właściwego rozszerzenia i zainstalowanie go. W oknie po prawej stronie znajdują się informacje dotyczące wybranego rozszerzenia – warto się z nimi zapoznać. W ten sposób dodajemy pozostałe rozszerzenia: Language Support for Java(TM) by Red Hat (znajduje się niżej na liście rozszerzeń) i Maven for Java – wystarczy wpisać słowo Maven w polu wyszukiwania.

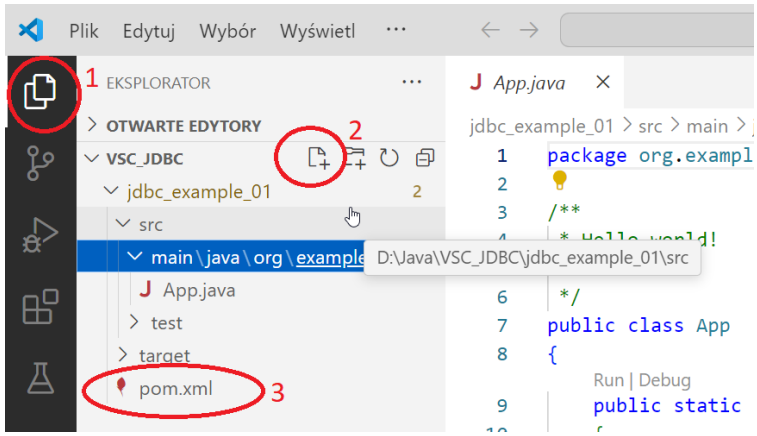


Rysunek 10.5: Visual Studio Code – instalowanie rozszerzeń

W celu utworzenia projektu należy: z menu „Wyświetl” wybrać polecenie „Paleta poleceń” (`Ctlr+Shift+P`), w oknie poleceń wpisać słowo Maven, z listy podpowiedzi wybrać „Maven:New Project...”. Następnie wybieramy archetyp. W naszym przypadku może to być „maven-archetype-quickstart”, podobnie jak w przypadku projektu w IntelliJ. W kolejnych krokach podajemy: numer wersji (np. 1.1), identyfikator grupy (np. „org.example”), identyfikator artefaktu, który stanowi zarazem nazwę projektu, np. „jdbc_example_01” i wskazać nazwę katalogu, w którym zostanie zapisany utworzony projekt. W trybie interaktywnym, w terminalu należy jeszcze podać numer wersji projektu. Jeżeli chcemy użyć wartości domyślnych, to wystarczy wcisnąć Enter i po zakończeniu tworzenia projektu wcisnąć dowolny klawisz w celu zamknięcia terminala.

Maven utworzy strukturę katalogów oraz plik „pom.xml” (rys. 10.6). Dostęp do struktury katalogów projektu mamy po kliknięciu na przycisk Exploratora (`Ctlr+Shift+E`), który znajduje się na pasku narzędzi po lewej stronie aplikacji, na górze (rys. 10.6 (1)). Teraz w podkatalogu *..\scr\java\org\example*, wystarczy umieścić pliki z listingów 10.2 i 10.3 lub utworzyć nowe pliki Javy po wciśnięciu przycisku służącego do dodania nowego pliku (rys. 10.6 (2)) i wpisać do nich kod.

Pozostaje jeszcze dodać do pliku pom.xml zależność z listingu listing 10.1. Po wprowadzeniu zmian pojawi się okienko dialogowe z pytaniem o synchronizację konfiguracji. Należy zatwierdzić zmiany. Program jest gotowy do kompilacji i uruchomienia.



Rysunek 10.6: Visual Studio Code – widok projektu

10.5. Praca z poleceniami SQL

Do wykonywania poleceń SQL potrzebny jest obiekt typu `Statement`. Można go uzyskać przy pomocy obiektu typu `Connection`, który z kolei jest zwracany przez metodę `getConnection` klasy `DriverManager` i reprezentuje połączenie z konkretną bazą danych.

```
Connection connection = DriverManager.getConnection(URL, user, password);
Statement statement = connection.createStatement();
```

Przy pomocy obiektu klasy `Statement` można wykonywać polecenia w języku SQL. Polecenie można zapisać w postaci łańcucha znaków:

```
String command = "UPDATE Cars" + " SET Price = Price + 1000"
                        + " WHERE Brand NOT LIKE '%Fiat%' ";
```

a następnie użyć w metodzie `executeUpdate` klasy `Statement`:

```
int changedRecords = stat.executeUpdate(commad);
```

Metoda zwraca liczbę zmodyfikowanych rekordów lub -1, gdy nie doszło do modyfikacji (wartość -1 zwracana jest również w przypadku poleceń, które nie modyfikują danych). W tym przykładzie będzie to liczba samochodów, którym podwyższono cenę.

Metodę `executeUpdate` stosuje się do wykonywania takich poleceń SQL, jak `INSERT`, `UPDATE` i `DELETE` oraz poleceń dotyczących definiowania danych: `CREATE TABLE`, `DROP TABLE`.

Do wykonania poleceń, które zwracają dane, np. `SELECT` używana jest metoda `executeQuery`. Można również skorzystać z metody `execute`, która umożliwia wykonanie dowolnego polecenia języka SQL. Zazwyczaj wykorzystuje się ją do wykonywania poleceń wprowadzanych przez użytkownika w trakcie działania aplikacji.

Metoda `executeQuery` zwraca obiekt typu `ResultSet`, który służy do odczytania kolejnych rekordów wyniku polecenia.

```
ResultSet results = statement.executeQuery("SELECT * FROM Cars");
```

Przeglądanie wyników odbywa się zazwyczaj w pętli:

```
while(results.next()) {
   // operacja na pojedynczym wierszu
}
```

Dostęp do pierwszego wiersza z tabeli wyników jest możliwy po wykonaniu metody `next`. Metoda ta zwraca `null` po dojściu do końca tabeli.

Przeglądając zawartość wiersza tabeli, można użyć wielu metod udostępniających zawartość poszczególnych pól, np.:

```
String name = results.getString(1)
double price = rs.getDouble("Price");
```

Istnieją metody dostępu odpowiadające każdemu z typów danych języka Java, na przykład `getString` czy `getDouble`. Są dwie wersje, jedna o parametrze numerycznym i druga o parametrze w postaci łańcucha znaków – nazwie kolumny tabeli w bazie danych. Używając parametru numerycznego, odwołujemy się do kolumny o danym numerze. Na przykład `results.getString(1)` zwraca wartość pierwszej kolumny dla danego wiersza. (**Ważne**: kolumny są numerowane od 1, a nie od 0.)

Metoda `execute` wykonuje polecenie języka SQL i zwraca wartość `true`, jeżeli jego wynikiem jest zbiór rekordów. Aby uzyskać wyniki, należy użyć metody `getResultSet` lub `getUpdateCount`.

Dla obiektu typu `Connection` można utworzyć wiele obiektów typu `Statement`. Ten sam obiekt typu `Statement` można użyć wiele razy do wykonania różnych poleceń języka SQL. Należy pamiętać o tym, że obiekt ten dysponuje co najwyżej jednym obiektem typu `ResultSet`. Jeśli w programie wykonujemy wiele zapytań, których wyniki musimy

później przetwarzać równocześnie, to należy utworzyć wiele obiektów typu `Statement`. Są sterowniki baz danych, np. Microsoft SQL Server, który dopuszcza tylko jeden aktywny obiekt typu `Statement`. Aby uzyskać informacje o maksymalnej liczbie aktywnych obiektów typu `Statement` przez danych sterownik, można użyć metody `getMaxStatement` interfejsu `DatabaseMetaData`.

Po zakończeniu korzystania z obiektów typu `ResultSet`, `Statement` czy `Connection` należy wywołać metodę `close`. Metoda `close` obiektu `Statement` zamyka związany z nim zbiór wyników, o ile jest otwarty. Analogicznie metoda `close` klasy `Connection` zamyka wszystkie obiekty poleceń związane z danym połączeniem.

W przypadku korzystania z połączeń krótkotrwałych, wystarczy zagwarantować, żeby obiekt połączenia nie pozostawał otwarty. Wówczas należy umieścić wywołanie metody `close` w bloku `finally` – listing 10.4.

LISTING 10.4: *Szkielet kodu dla krótkotrwałych połączeń z bazą danych*

```
try {
   Connection connection = ...;
   try {
     Statement statement = connection.createStatement();
     ResultSet result = statement.executeQuery(query);
     // przetwarzanie wyniku zapytania
   }
   finally {
     connection.close()
   }
}
catch { // obsługa wyjątków
}
```

10.6. Wyjątki SQLException

W trakcie wykonywania pojedynczego polecenia SQL może wystąpić wiele błędów, wówczas zostaje wygenerowany wyjątek. Dla wyjątku `SQLException` może powstać łańcuch obiektów `SQLException`, które można pobrać z wykorzystanie metody `getNextException`, a każdy wyjątek z łańcucha odpowiada jednemu błędowi. Od wersji 6 Javy można skorzystać z iteratora `Iterator<Throwable>`, który umożliwia przegląd wszystkich wyjątków z użyciem jednej pętli:

```
for (Throwable e : sqlException)  {
  // operacje na obiekcie e, np. e.printStackTrace();
}
```

Należy zwrócić uwagę, że iterator, podobnie jak metoda `getCause` umożliwia dostęp do obiektów typu `Throwable`, natomiast metod `getNextException` do obiektów klasy `SQLException`.

Dokładniejsze informacje o wyjątku można otrzymać poprzez metody `getSQLState` i `getErrorCode`. Metoda `getMessage` zwraca opis wyjątku. Metoda `getSQLState` zwraca informację w postaci pięcioznakowego kodu o błędzie lub o ostrzeżeniu. W tym kodzie dwa pierwsze znaki reprezentują klasę błędu, natomiast pozostałe trzy znaki podklasę błędu, która daje bardziej szczegółowe informację o problemie. Na przykład kod zaczynający się od znaków „80" oznacza problemy z połączeniem z bazą (ang. *Connection Exception*), a trzy jego podklasy następujące błędy:

- "08001": SQL-client unable to establish SQL-connection.
- "08003": Connection does not exist.
- "08006": Connection failure.

W przypadku braku połączenia z bazą danych możemy otrzymać następujące informacje o problemie:

```
SQLState: 08001
Error Code: 0
Message: No suitable driver found for
              jdbc:postgresql://localhost:5432/mydatabase
```

Sterowniki baz danych mogą również raportować ostrzeżenia. Służy do tego klasa `SQLWarning`, która jest klasą pochodną `SQLException`. Informacje dotyczące wystąpienia ostrzeżeń można pobrać wywołując metody `getSQLState` i `getErrorCode`. Podobnie jak wyjątki SQL, również ostrzeżenia tworzą łańcuchy. Aby pobrać wszystkie ostrzeżenia, można skorzystać z pętli:

```
1 SQLWarning w = stat.getWarning();
2 while (w != null) {
3   // operacje na obiekcie w
4   w = w.nextWarning();
5 }
```

W celu uniknięcia powielania kodu zajmującego się obsługą wyjątków związanych z bazami danych napiszemy metodę `main` (listing 10.5) w ten sposób, aby to ona kompleksowo zajmowała się obsługą wyjątków. Natomiast metody prezentujące kolejne przykłady korzystania z bazy danych będą deklarowały możliwość zgłoszenia wyjątku `SQLException` – na przykład metoda `createTable` z listingu 10.5.

W kolejnych przykładach wystarczy zastąpić metodę `createTable` nową metodą lub umieścić wywołanie nowej metody w kolejnej linii, za metodą `crateTable`. Oczywiście ważna jest kolejność wykonywania operacji na bazie danych. Najpierw powinniśmy utwo-

rzyć tabelę, później wypełnić ją danymi, a na końcu wykonać zapytanie związane z tą tabelą. Jeżeli na przykład drugi raz uruchomimy program, który tworzy tabelę, to zapewne pojawi się wyjątek informujący, że taka tabela już istnieje. Natomiast na utworzonych i wypełnionych danymi tabelach można wielokrotnie wykonywać różne zapytania.

LISTING 10.5: *Kod programu z metodą* `main`*, która zajmuje się obsługą wyjątków typu* `SQLException`

```
public class Example_02 {
    public static void main( String[] args ) {
        Connection connection = null;
        try {
            createTable(connection);
        }
        catch (SQLException e) {
            System.err.println("SQL State: " + e.getSQLState());
            System.err.println("Error Code: " + e.getErrorCode());
            System.err.println(e.getMessage());
            Throwable t = e.getCause();
            while (t != null) {
                System.out.println("Cause: " + t);
                t = t.getCause();
            }
        }
        finally {
            try {
              if (connection != null) {
                DBConnector.close(connection);
              }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
        public static void createTable(Connection connection)
                        throws SQLException {
        // ...
    }}
```

10.7. Przykład – tworzenie tabeli w bazie danych

Aby utworzyć tabelę w bazie danych, należy użyć metody `executeUpdate` klasy `Statement`, przekazując jako parametr polecenie SQL służące do utworzenia konkretnej tabeli (listing 10.6 wiersz 7). Metoda `executeUpdate` zwraca 0 w przypadku poleceń SQL, które nie zwracają żadnej wartości lub liczbę zaktualizowanych wierszy, w przypadku poleceń SQL, które wprowadzają zmiany.

W wierszu 5 znajduje się wywołanie polecenia SQL, które bez żadnego ostrzeżenia usuwa, o ile istnieje, tabelę `t_klient` niezależenie od tego, czy tabela jest pusta, czy wypełniona danymi.

Listing 10.6: *Metoda* `createTable` *klasy* `Example_2`*, która tworzy tabelę w bazie danych*

```
1  public static void createTable(Connection connection)
2                              throws SQLException {
3          connection = DBConnector.connect();
4          Statement statement = connection.createStatement();
5          int res = statement.executeUpdate(
6                        "DROP TABLE IF EXISTS t_klient");
7          String query = "CREATE TABLE t_klient(" +
8            "id_klienta int PRIMARY KEY," +
9            "imie varchar(30)," +
10           "nazwisko varchar(40)," +
11           "ulica varchar(30)," +
12           "miasto varchar(30)," +
13           "wojewodztwo varchar(30)," +
14           "kod char(6) CHECK(kod ~ ('^\\d{2}-\\d{3}$'))," +
15           "telefon char(11) CHECK(telefon ~ ('^\\d{3}-\\d{3}-\\d
     {3}$')));";
16         res = statement.executeUpdate(query);
17         statement.close();
18     }
```

10.8. Przykład – wstawianie danych do tabeli

Rzadko zdarza się, aby w kodzie Javy umieszczać dane przeznaczone do wypełnienia tabeli, szczególnie jeżeli tych danych jest dużo. W prezentowanym przykładzie dane będą znajdowały się w pliku tekstowych (listing 10.7), który zawiera polecenia SQL służące do wstawienia rekordów do tabeli `t_klient`.

Listing 10.7: *Plik* `t_klient.sql`*, zawierający polecenia SQL służące do wypełnienia tabeli* `t_klient` *przykładowymi danymi*

```
INSERT INTO t_klient VALUES('1', 'Jan', 'Kowalski',
 'Akacjowa', 'Warszawa', 'mazowieckie', '00-950', '502-501-501');
INSERT INTO t_klient VALUES('2', 'Antoni', 'Kwiatkowski',
 '1 Maja', 'Poznań', 'wielkopolskie', '60-002', '503-533-533');
INSERT INTO t_klient VALUES('3', 'Maja', 'Smith',
 'Szczytowa', 'Łódź', 'łódzkie', '70-445', '501-233-453');
INSERT INTO t_klient VALUES('4', 'Hernyk', 'Maciąg', 'Leśna',
 'Iława', 'warmińsko-mazurskie', '14-200', '607-113-783');
INSERT INTO t_klient VALUES('5', 'Michał', 'Gołąb',
 'Ikara', 'Siewierz', 'śląskie', '42-470', '606-552-983');
INSERT INTO t_klient VALUES('6', 'Anna', 'Basińska',
 'Lipowa', 'Sopot', 'pomorskie', '80-336', '505-236-903');
INSERT INTO t_klient VALUES('7', 'Dariusz', 'Wałek',
 '3 Maja', 'Poronin', 'małopolskie', '34-425', '602-003-677');
INSERT INTO t_klient VALUES('8', 'Telimena', 'Walewska',
 'Rocha', 'Gdańsk', 'pomorskie', '80-002', '603-535-517');
INSERT INTO t_klient VALUES('9', 'Joanna', 'Kowal',
 '1 Maja', 'Sosnowiec', 'śląskie', '34-112', '703-522-636');
INSERT INTO t_klient VALUES('10', 'Grzegorz', 'Kwiatek',
 'Ludowa', 'Ełk', 'warmińsko-mazurskie', '19-300', '701-583-983');
```

Działanie metody `insertDate` (listing 10.8) sprowadza się do odczytania kolejnych wierszy z pliku tekstowego, które stanowią polecenia SQL i wykonanie ich z wykorzystaniem metody `execute`.

Zwracamy uwagę, że łatwo można zmodyfikować tę metodę tak, aby nazwa pliku z poleceniami SQL była przekazywana jako parametr. Wówczas można napisać prosty program, w którym z linii poleceń podawałoby się nazwę pliku z poleceniami SQL. W ten sposób można szybko utworzyć wiele tabel bazy danych oraz wypełnić je danymi.

Listing 10.8: *Metoda* `insertDate` *klasy* `Example_2`*, która wypełnia tabelę* `t_klient` *przykładowyi danymi*

```
public static void insertDate(Connection connection) throws
    SQLException {
   connection = DBConnector.connect();
   try {
      BufferedReader file = new BufferedReader(new FileReader("
    t_klient.sql"));
      Statement statement = connection.createStatement();
      String query;
```

```
7       while ((query = file.readLine()) != null) {
8          boolean result = statement.execute(query);
9       }
10      statement.close();
11      file.close();
12   } catch (FileNotFoundException e) {
13       System.err.println(e.getMessage());
14   } catch (IOException ex) {
15      System.err.println(ex.getMessage());
16   }}
```

10.9. Przykład – wykonywanie zapytań

Wyszukując interesujące nas dane w bazie danych, możemy skorzystać ze zwykłego zapytania SQL lub użyć zapytania przygotowawczego.

Zapytanie przygotowawcze umożliwia tworzenie zapytań SQL z parametrami, co pomaga w ochronie przed atakami SQL Injection. Zamiast umożliwiać użytkownikowi tworzenie kompletnych zapytań, dajemy mu możliwość podania parametrów dla zapytań przygotowanych przez nas. Drugą zaletą tych zapytań jest większa efektywność dzięki możliwości buforowania zapytań przez silnik bazy danych.

Aby móc skorzystać z zapytań przygotowawczych, należy utworzyć obiekt typu `PreparedStatement`, do którego przekazujemy zapytanie SQL, np.:

```
String sqlQuery = "SELECT wydawca, cena FROM t_ksiazka " +
                          "WHERE wydawca = ?" + " AND cena > ?";
PreparedStatement preparedStatement =
                           connection.prepareStatement(sqlQuery);
```

W zapytaniu miejsce parametru jest oznaczane znakiem zapytania. Wartość parametru podajemy przy użyciu metod „setterów", które umożliwiają przekazanie parametrów określonego typu, np. `setInt`, `setString`. Metody te posiadają dwa parametry. Pierwszy parametr metody określa zmienną (numer znaku ?), której chcemy nadać wartość. Wartość 1 wskazuje na pierwsze wystąpienie znaku ? w poleceniu SQL. Drugi parametr metody określa wartość, którą nadajemy zmiennej. W przykładzie pierwsze wystąpienie znaku ? określa nazwę wydawcy, czyli wartość typu `String`. Wartość parametru zapytania zostanie ustalona przy pomocy instrukcji `preparedStatement`, np.:

```
preparedStatement.setString(1, "Addison-Wesley");
```

Natomiast drugie wystąpienie znaku ? określa minimalną cenę książki, np.:

```
preparedStatement.setDouble(2, 50);
```

Metoda `selectPublisher` przedstawiona na listingu 10.9 umożliwia utworzenie i wykonanie zapytania, które służy do wyszukania w tabeli książek wydanych przez wskazanego wydawcę oraz kosztujących więcej niż podana kwota. W wyniku zapytania otrzymujemy na ekranie informację o książkach spełniających ustalone przez użytkownika kryteria. Do wykonania zapytania używamy metody `executeQuery` z klasy `PreparedStatement`.

LISTING 10.9: *Metoda* `selectPublisher` *klasy* `Example_2` *wypisująca na ekranie książki wskazanego wydawcy, które kosztują więcej niż wskazana kwota*

```
public static void selectPublisher(Connection connection,
          String name, double price) throws SQLException {
  connection = DBConnector.connect();
  Statement statement = connection.createStatement();
  String sqlQuery = "SELECT wydawca, cena FROM t_ksiazka" +
                           " WHERE wydawca = ?" +
                            " AND cena > ?";
  PreparedStatement preparedStatement =
                  connection.prepareStatement(sqlQuery);
  preparedStatement.setString(1, name);
  preparedStatement.setDouble(2, price);
  ResultSet results = preparedStatement.executeQuery();
  while ( results.next() ) {
    System.out.print(results.getString("wydawca") + " - ");
    System.out.println(results.getDouble("cena") + ". ");
  }
  statement.close();
}
```

10.10. Przykład – tworzenie aplikacji z GUI

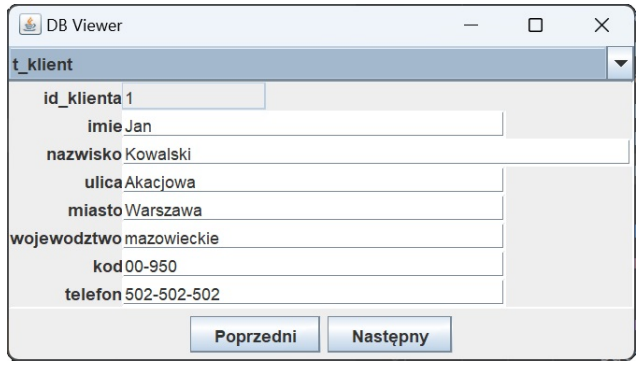
W tym przykładzie zostanie przestawiony sposób wykorzystania prostego graficznego interfejsu użytkownika do wyświetlania danych pochodzących z bazy danych.

Aplikacja będzie umożliwiała przeglądanie danych pochodzących z poszczególnych tabel (rys. 10.7). Na górze okna znajduje się lista rozwijana, z której użytkownik może wybrać dowolną tabelę z podłączonej bazy danych. Po wybraniu tabeli, na panelu pojawią się nazwy kolumn oraz wartości pierwszego rekordu. Przy pomocy klawiszy „Poprzedni", „Następny" użytkownik może przeglądać wszystkie rekordy wybranej tabeli. Po wybraniu nowej tabeli, na panelu pojawią się nazwy kolumn pochodzące z nowej tabeli.

Na listingach 10.10-10.14 znajdują się najistotniejsze fragmenty aplikacji. Kod całego programu znajduje się w „Dodatku B".

Do stworzenia GUI panelu będą potrzebne następujące komponenty:

```
private JButton previousButton;
private JButton nextButton;
private DBPanel dataPanel;
private JScrollPane scrollPane;
private final JComboBox<String> tableNames;
```



Rysunek 10.7: Okno aplikacji

Są to przyciski (`JButton`) umożliwiające przejście do kolejnego lub poprzedniego rekordu, lista rozwijana (`JComboBox`) służąca do wyboru tabeli, panel z możliwością przewijania zawartości (`JScrollPane`). Umożliwia przewijanie zawartości, gdy ta nie mieści się w oknie. Wewnątrz tego panela zostanie umieszczony panel (`DBPanel`), który wyświetla rekordy z bazy danych.

Klasa `DBPanel` reprezentuje klasę dziedziczącą po klasie `JPanel` – komponent, na którym wyświetlane są rekordy z wybranej tabeli.

Na listingu 10.10 została umieszczona definicja konstruktora klasy `DBPanel`. Do konstruktora zostaje przekazany parametr typu `RowSet` zawierający wynik zapytania SQL pobierającego wszystkie elementy wybranej przez użytkownika tabeli.

Klasa `RowSet` umożliwia przechowywanie zbioru danych, podobnie jak klasa `ResultSet`, ale jest bardziej elastyczna i ma większe możliwości, np. może pracować w trybie bez połączenia z bazą danych, można łatwo „przewijać" dane w tył i przód, czy je aktualizować.

Do rozmieszczenia komponentów na panelu został użyty menadżer rozkładu `GridBagLayout` – linia 3. Ustawia on komponenty na dwuwymiarowej siatce, ale ma dodatkowe możliwości, np. dokładne wskazanie położenia komponentu (pola `gbc.gridx` i `gbc.gridy`), określenie rozmiaru (`gbc.gridwidth` i `gbc.gridheight`), czy wyrównania zawartości komponentu (`gbc.anchor`). Do określenia parametrów rozmieszczenia komponentów umieszczanych na tym rozkładzie służy obiekt klasy `GridBagConstraints`.

Zawartość tabeli została wyświetlona na panelu w dwóch kolumnach. W pierwszej, użyto etykiet do wyświetlenia nazw kolumn tabeli. Napisy zostały wyrównane do strony prawej (linia 12), natomiast do wyświetlenia danych użyto pól tekstowych, których zawartość wyrównana jest do lewej strony (linia 19). Dzięki użyciu pól tekstowych użytkownik może zaznaczyć myszką ich zawartość i ją skopiować ale nie może ich modyfikować – linia 16. W linii 7 mamy odczytanie metadanych tabeli, natomiast już w linii 8 jest pierwsze odwołanie do tych danych – odczyt liczby kolumn tabeli. W każdym przebiegu pętli dodawana jest do panelu para komponentów reprezentujących nazwę kolumny tabeli (linia 13 i 17) oraz pole tekstowe, w którym później zostanie umieszczona wartość pierwszego rekordu tej kolumny. Odczyt nazwy kolumny jest w linii 10. W przypadku pola tekstowego, w konstruktorze tworzone jest pole tekstowe niewypełnione danymi. Jest tylko pobierany rozmiar danych (np. długość, napisu, czy liczba znaków dla liczby) i ustalany rozmiar tego pola tak, aby cała była wyświetlona na ekranie – lina 14 i 15.

Metoda `add` dodaje komponent na panel, drugi parametr określa ogranicznik. Pole `fields` jest listą tablicową typu `JTextField`, w której przechowywane są pola tekstowe zawierające poszczególne nazwy kolumn wybranej tabeli bazy danych. Wszystkie utworzone w pętli pola tekstowe zostają także dodane do listy tablicowej `fields` (linia 17).

LISTING 10.10: *Konstruktor klasy* `DBPanel`

```
1  public DBPanel(RowSet rs) throws SQLException {
2    fields = new ArrayList<>();
3    setLayout(new GridBagLayout());
4    GridBagConstraints gbc = new GridBagConstraints();
5    gbc.gridwidth = 1;
6    gbc.gridheight = 1;
7    ResultSetMetaData rsmd = rs.getMetaData();
8    for (int i = 1; i <= rsmd.getColumnCount(); i++) {
9      gbc.gridy = i - 1;
10     String columnName = rsmd.getColumnLabel(i);
11     gbc.gridx = 0;
12     gbc.anchor = GridBagConstraints.EAST;
13     add(new JLabel(columnName + "  "), gbc);
14     int columnWidth = rsmd.getColumnDisplaySize(i);
15     var tb = new JTextField(columnWidth);
16     tb.setEditable(false);
17     fields.add(tb);
18     gbc.gridx = 1;
19     gbc.anchor = GridBagConstraints.WEST;
20      add(tb, gbc);
21 }
```

Ponieważ aplikacja wyświetla również informacje o nazwach tabel, czy nazwach kolumn, to skądś musi uzyskać te informacje. Tego typu dane nazywane są metadanymi i w programie Java można je odczytać z bazy danych z wykorzystaniem obiektu klasy. Klasa `DatabaseMetaData` służy do odczytu informacji o bazie danych, natomiast klasa `ResultSetMetaData` umożliwia uzyskanie informacji o zbiorze wyników. Gdy w rezultacie wykonania zapytania otrzymamy zbiór wyników, można również uzyskać informację o liczbie kolumn oraz nazwie, typie i rozmiarze każdej z kolumn. Pętla przeglądająca takie metadane może wyglądać następująco:

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData metaData = rs.getMetaData();
for (int i = 1; i <= metaData.getColumnCount(); i++) {
  String columnName = metaData.getColumnLabel(i);
  int columnWidth = metaData.getColumnDisplaySize(i);
. . .
}
```

Na listingu 10.11 przedstawiona jest metoda służąca do wyświetlenia na panelu pojedynczego rekordu danych. Jest ona wykorzystywana zarówno do wyświetlenia pierwszego rekordu, zaraz po wyborze tabeli, a także przy przechodzeniu do poprzedniego lub następnego rekordu.

Rekord danych przeznaczony do wyświetlenia w panelu jest przekazywany jako parametr metody. W metodzie wewnątrz pętli, która przebiega przez wszystkie elementy listy tablicowej `fields` następuje pobranie kolejnego pola tekstowego (linia 6) i wpisanie do niego tekstu pobranego z rekordu danych (linia 5) reprezentowanego przez zmienną `rs`. Wyświetlenie napisu następuje w linii 7.

LISTING 10.11: *Metoda* `showRow` *klasy* `DBPanel` *wyświetlająca rekord danych na panelu*

```
1  public void showRow(ResultSet rs) {
2    try {
3      if (rs == null) return;
4      for (int i = 1; i <= fields.size(); i++) {
5        String field = rs == null ? "" : rs.getString(i);
6         JTextField tb = fields.get(i - 1);
7         tb.setText(field);
8       }
9    }
10   catch (SQLException ex) {
11     MySQLExceptionInfo.print(ex);
12   }
13 }
```

W konstruktorze klasy publicznej (listing 10.12) znajdują się instrukcje odpowiedzialne za:

- utworzenie komponentów, które zostaną wyświetlone w okienku aplikacji;
- utworzenie i przypisanie do wybranych komponentów (np. przycisków, listy rozwijanej) odpowiednich słuchaczy, w celu iteracji z użytkownikiem;
- przygotowanie danych, które pojawią się w komponentach widocznych po uruchomieniu aplikacji, np. tabele w liście rozwijanej.

W linii 2 został utworzony obiekt klasy `JComboBox`, który powinien zostać wypełniony nazwami tabel z bazy danych. W linii 4 następuje połączenie aplikacji z bazą danych i odczytanie w linii 4 i 5 metadanych o tabelach. W kolejnych instrukcjach, w pęli `while` następuje odczyt kolejnych nazw tabel i zapisanie ich nazw na liście rozwijanej `tableNames`. Metoda `getTable` posiada cztery parametry: pierwszy określa katalog, w którym wyszukiwane są tabele, drugi określa wzorzec schematu (grupy tabel), trzeci wzorzec nazwy tabeli (np. UJD%, aby wyszukać tabele zaczynające się od nazwy UJD), czwarty określa tablicę łańcuchów znaków, która reprezentuje typy obiektów, np. `TABLE`, `SYSTEM TABLE`). Wartości `null` powodują przyjęcie wartości domyślnej przez metodę.

Po użyciu obiektu typu `ResultSet` należy wywołać metodę `close`. Powoduje to zwolnienie zasobów bazy danych. Wykonanie metody `close` zagwarantuje użycie konstrukcji `try-finally`.

W wierszu 17 do obiektu `tableNames` dodany jest słuchacz akcji, który powinien reagować na wybór pozycji z listy przez użytkownika. W naszym przypadku spowoduje to wywołanie metody `showTable`, do której zostanie przekazana nazwa tablicy (metoda `tableName.getSelectedItem` zwraca wybraną pozycję z listy rozwijanej) oraz obiekt reprezentujący połączenie z bazą danych. W linii 22 obiekt reprezentujący listę rozwijaną zostaje dodany do okienka aplikacji.

Kolejne instrukcje (linie 23-32) reprezentują obsługę zdarzenia polegającego na zamknięciu okienka aplikacji. Obsługa tego zdarzenia polega na zakończeniu połączenia z bazą danych. Następnie tworzony jest obiekt reprezentujący panel, na którym zostaną umieszczone przyciski oraz same przyciski. Do przycisków dodawani są słuchacze akcji. Obsługa przycisku „Poprzedni" polega na wywołaniu metody `showPreviousRowj`, a przycisku „Następny" na wywołaniu metody `showNextRow`. W ostatniej instrukcji konstruktora następuje sprawdzenie, czy na liście rozwijanej są jakieś pozycje. Jeśli tak, to w okienku aplikacji wyświetlany jest pierwszy rekord z pierwszej tabeli

Listing 10.12: *Konsturktor* `MainFrame` *klasy publicznej*

```
1 public MainFrame() {
2   tableNames = new JComboBox<String>();
3   try {
4     connection = DBConnector.connect();
```

```
5     DatabaseMetaData meta = connection.getMetaData();
6     ResultSet mrs = meta.getTables(null, null, null, new String[]{
      "TABLE"});
7     try {
8       while (mrs.next())
9         tableNames.addItem(mrs.getString(3));
10    } finally {
11      mrs.close();
12    }
13  }
14  catch (SQLException ex) {
15    MySQLExceptionInfo.print(ex);
16  }
17
18  tableNames.addActionListener( new ActionListener() {
19    @Override public void actionPerformed(ActionEvent e) {
20       showTable( (String) tableNames.getSelectedItem(),
      connection);
21   }} );
22
23   add(tableNames, BorderLayout.NORTH);
24
25   addWindowListener(new WindowAdapter()  {
26     public void windowClosing(WindowEvent event)  {
27       try {
28         if (connection != null) connection.close();
29       }
30       catch (SQLException ex) {
31         MySQLExceptionInfo.print(ex);
32       }
33     }
34   });
35
36   JPanel buttonPanel = new JPanel();
37   add(buttonPanel, BorderLayout.SOUTH);
38   previousButton = new JButton("Poprzedni");
39   previousButton.addActionListener( new ActionListener() {
40     @Override public void actionPerformed(ActionEvent e) {
41       showPreviousRow();
42     }
43   });
44   buttonPanel.add(previousButton);
```

```
45    nextButton = new JButton("Następny");
46    nextButton.addActionListener( new ActionListener() {
47      @Override public void actionPerformed(ActionEvent e) {
48        showNextRow();
49      }
50    });
51    buttonPanel.add(nextButton);
52    if (tableNames.getItemCount() > 0)
53      showTable(tableNames.getItemAt(0), connection);
54  }
```

Na listingu 10.13 znajduje się metoda `showTable` służąca do wyświetlenia na ekranie pojedynczego rekordu danych znajdującego się w tablicy, której nazwa przekazana jest jako pierwszy parametr. Metoda wykorzystywana jest każdorazowo przy wyborze nowej tablicy na liście rozwijanej. Wiąże się to z usunięciem z okienka aplikacji aktualnego panelu i zastąpieniu go nowym, który jest dopasowany do wyświetlenia danych pochodzących z nowej tablicy. Na początku metody następuje połączenie z bazą danych oraz pobranie wszystkich danych z tablicy o wskazanej przez użytkownika nazwie.

W wierszu 4 utworzono obiekt implementujący interfejs `RowSetFactory`, który służy do wygenerowania zbioru wyników typu `CachedRawSet`. Ten typ umożliwia działanie w trybie odłączonym od bazy danych, przechowywanie wyników w pamięci RAM. Obiekt `crs` jest polem klasy `MainFrame` typu `CachedRawSet`. Instrukcja `crs.populate(result);` wypełnia zbiór danych danymi, natomiast instrukcja `crs.setTableName(tableName)` łączy dane ze zbioru z nazwą tabeli. Następnie panel wyświetlający uprzednio wybraną tablicę jest usuwany (o ile istnieje) i tworzony jest nowy panel, w którym pojawią się dane z nowo wybranej tablicy. Prościej jest utworzyć nowy panel, niż modyfikować istniejący. Nowe panele dodawane są do okienka aplikacji, zastępując poprzednie komponenty (wiersze 9-11). Metoda `pack` powoduje dopasowanie rozmiaru panelu do zawartości. Na koniec wywoływana jest metoda `showNextRow`, aby wyświetlić pierwszy rekord.

Listing 10.13: *Metoda* `showTable` *klasy* `MainFrame` *wyświetlająca rekord danych na panelu*

```
1  public void showTable(String tableName, Connection conn) {
2    try (Statement stat = conn.createStatement();
3         ResultSet result = stat.executeQuery("SELECT * FROM " +
       tableName))  {
4      RowSetFactory factory = RowSetProvider.newFactory();
5      crs = factory.createCachedRowSet();
6      crs.setTableName(tableName);
7      crs.populate(result);
8      if (scrollPane != null) remove(scrollPane);
```

```
      dataPanel = new DBPanel(crs);
      scrollPane = new JScrollPane(dataPanel);
      add(scrollPane, BorderLayout.CENTER);
      pack();
      showNextRow();
    }
    catch (SQLException ex) {
      MySQLExceptionInfo.print(ex);
    }
}
```

Na listingu 10.14 znajduje się metoda `showNextRow` klasy `MainFrame` wyświetlająca kolejny rekord danych na panelu. Na początku w instrukcji warunkowej następuje sprawdzenie, czy obiekt reprezentujący zbiór danych nie przyjmuje wartości `null` lub na ekranie wyświetlony jest ostatni rekord. Jeżeli tak, to metoda kończy działanie. W przeciwnym razie wywołana jest metoda `next`, która przesuwa kursor w zbiorze danych na kolejny rekord i następuje wywołanie metody `showRow` z klasy `DBPanek` z przekazanym, nowym rekordem danych do wyświetlenia na panelu.

Listing 10.14: *Metoda* `showNextRow` *klasy* `MainFrame` *wyświetlająca kolejny rekord danych na panelu*

```
public void showNextRow() {
        try {
            if (crs == null || crs.isLast()) return;
            crs.next();
            dataPanel.showRow(crs);
        }
        catch (SQLException ex) {
            MySQLExceptionInfo.print(ex);
        }
}
```

10.11. Zadania do samodzielnego rozwiązania

1. Napisz program, który wykonuje polecenia SQL zapisane w pliku tekstowym. Nazwę pliku tekstowego użytkownik przekazuje do programu w linii poleceń.
2. Zmodyfikuj metodę `createTable` z listingu 10.6 w ten sposób, aby przyjmowała dwa parametry. Drugim parametrem byłoby polecenie SQL służące do tworzenia tabel. Utwórz przy pomocy tej metody tabelę `t_ksiazka` opisującą książki i wypełnij ją danymi z listingu 10.15.

3. Napisz aplikację (rozwinięcie aplikacji z podrozdziału 10.10), w której można wyświetlić poprzedni rekord danych oraz przejść do pierwszego lub ostatniego rekordu zbioru wyników, klikając odpowiednio na przycisku „Pierwszy” i „Ostatni”.
4. Napisz aplikację (rozwinięcie aplikacji z podrozdziału 10.10, w której można dodawać, usuwać i edytować rekordy bazy danych.

LISTING 10.15: *Plik* `t_ksiazka.sql`*, zawierający polecenia SQL służące do wypełnienia tabeli* `t_ksiazka` *przykładowymi danymi*

```
INSERT INTO t_ksiazka VALUES ('1', 'C.J. Date',
   'A Guide to the SQL Standard', 'Addison-Wesley' , 6.07);
INSERT INTO t_ksiazka VALUES ('2', 'B. Schneier',
   'Applied Cryptography', 'Wiley', 46.66);
INSERT INTO t_ksiazka VALUES ('3', 'C. Stoll',
   'The Cuckoos Egg', 'Pocket books', 12.69);
INSERT INTO t_ksiazka VALUES ('4', 'E. Gamma et al.',
   'Design Patterns', 'Addison-Wesley', 50.68);
INSERT INTO t_ksiazka VALUES ('5', 'T. H. Cormen et al.',
   'Introduction to Algorithms', 'The MIT Press', 107.95);
INSERT INTO t_ksiazka VALUES ('6', 'D. Flanagan',
   'JavaScript: The Definitive Guide', 'OReilly Media', 57.95)
   ;
INSERT INTO t_ksiazka VALUES ('7', 'B. W. Kernighan',
   'The C Programming Language', 'Pearson', 55.99);
INSERT INTO t_ksiazka VALUES ('8', 'B. Stroustrup',
   'The C++ Programming Language', 'Addison-Wesley', 67.77);
INSERT INTO t_ksiazka VALUES ('9', 'E. S. Raymond',
   'The Cathedral and the Bazaar', 'OReilly Media', 61.54);
INSERT INTO t_ksiazka VALUES ('10', 'D. Kahn',
   'The Codebreakers', 'Scribner', 26.93);
INSERT INTO t_ksiazka VALUES ('11', 'F. Brooks Jr.',
   'The Mythical Man-Month', 'Addison-Wesley', 37.39);
INSERT INTO t_ksiazka VALUES ('12', 'T. Kidder ',
   'The Soul of a New Machine', 'Back Bay Books', 11.39);
```

Bibliografia

[1] Ferrari L., Pirozzi E., *Learn PostgreSQL - Second Edition*, Packt Publishing, 2023.

[2] Luzanov P., Rogov E., Levshin I. (translated by Mantrova L.), *POSTGRES: The First Experience*, Packt Publishing, 2023.

[3] Molinaro B., de Graaf R., *SQL Cookbook: Query Solutions and Techniques for All SQL Users, 2nd Edition*, O'Reilly Media, Inc, USA, 2021.

[4] Schönig Hans-Jürgen, *Mastering PostgreSQL 15. Advanced techniques to build and manage scalable, reliable, and fault-tolerant database applications - Fifth Edition (ebook)*, Packt Publishing, 2023.

[5] Shan J., Goldwasser M., Malik U., Johnston B., *SQL for Data Analyticd: Harness the power of SQL to extract insights for data*, 3rd Edition, Packt Publishing, 2022.

[6] Viescas J. L., Hernandez M. J., *SQL Queries for Mere Mortals: A Hands-On Guide to Data Manipulation in SQL*, Third Edition, Pearson Education, Inc, publishing as Addison Wesley, 2014.

[7] Horstmann C. S., *Core Java, Volume II – Advanced Features, 11th Edition*, Pearson Education, Inc, publishing as Prentice Hall, 2019.

[8] `https://jdbc.postgresql.org/`

[9] `https://www.postgresqltutorial.com/postgresql-jdbc/`

`https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`

Dodatek A

```
DROP DATABASE IF EXISTS ksiegarnia;
CREATE DATABASE ksiegarnia;
\c ksiegarnia

DROP TABLE IF EXISTS t_z_ksiazka CASCADE;
DROP TABLE IF EXISTS t_a_ksiazki CASCADE;
DROP TABLE IF EXISTS t_ksiazka CASCADE;
DROP TABLE IF EXISTS t_zamowienie CASCADE;
DROP TABLE IF EXISTS t_autor CASCADE;
DROP TABLE IF EXISTS t_wydawca CASCADE;
DROP TABLE IF EXISTS t_klient CASCADE;
DROP TABLE IF EXISTS t_dostawca CASCADE;

CREATE TABLE t_ksiazka(
  ISBN int2 PRIMARY KEY,
  tytul varchar(40) NOT NULL,
  wydawca int2,
  rok char(4) DEFAULT '2023' NOT NULL,
  oprawa char(6) CHECK(oprawa in ('miękka', 'twarda')),
  dostawca int2,
  cena decimal(4,2) CHECK(cena >= 0.0),
  ilosc int2 CHECK(ilosc >= 0),
  CHECK(rok ~ ('^\d{4}$'))
);

CREATE TABLE t_autor (
  id_autora int2 PRIMARY KEY,
  imie varchar(40),
  nazwisko varchar(40)
);
```

```
CREATE TABLE t_a_ksiazki (
  id_autora int2,
  ISBN int2
);

ALTER TABLE t_a_ksiazki
ADD CONSTRAINT klucz PRIMARY KEY (id_autora, ISBN);

ALTER TABLE t_a_ksiazki
ADD CONSTRAINT klucz_od_ksiazki FOREIGN KEY (ISBN)
REFERENCES t_ksiazka(ISBN) ON DELETE CASCADE;

ALTER TABLE t_a_ksiazki
ADD CONSTRAINT klucz_od_autora FOREIGN KEY (id_autora)
REFERENCES t_autor(id_autora) ON DELETE CASCADE ;

CREATE TABLE t_wydawca (
  id_wydawcy int2 PRIMARY KEY,
  nazwa varchar(10),
  adres varchar(40)
);

ALTER TABLE t_ksiazka
ADD CONSTRAINT klucz_od_wydawcy FOREIGN KEY (wydawca)
REFERENCES t_wydawca(id_wydawcy) ON DELETE CASCADE ;

CREATE TABLE t_zamowienie (
  id_zam int2 primary key,
  id_klienta int2,
  data_zam DATE DEFAULT current_date,
  karta int2,
  zrealizowane int2
);

CREATE TABLE t_z_ksiazka (
  id_zam int2,
  isbn int2,
  data_Wys DATE DEFAULT current_date,
  ilosc int2
);
```

```
CREATE TABLE t_klient(
  id_klienta int2 PRIMARY KEY,
  imie varchar(30),
  nazwisko varchar(40),
  ulica varchar(30),
  miasto varchar(30),
  wojewodztwo varchar(30),
  kod char(6) CHECK(kod ~ ('^\d{2}-\d{3}$')),
  telefon char(11) CHECK(telefon ~ ('^\d{3}-\d{3}-\d{3}$'))
);

ALTER TABLE t_zamowienie
ADD CONSTRAINT klucz_od_klienta FOREIGN KEY(id_klienta)
REFERENCES t_klient(id_klienta) ON DELETE CASCADE;

ALTER TABLE t_z_ksiazka
ADD CONSTRAINT klucz_od_zamowienia FOREIGN KEY(id_zam)
REFERENCES t_zamowienie(id_zam) ON DELETE CASCADE;

CREATE TABLE t_dostawca (
  id_dostawcy int2 PRIMARY KEY,
  nazwa varchar(30),
  ulica varchar(30),
  miejscowosc varchar(30),
  wojewodztwo varchar(40),
  kod varchar(6) CHECK(kod ~ ('^\d{2}-\d{3}$')),
  telefon varchar(11) CHECK(telefon ~ ('^\d{3}-\d{3}-\d{3}$'))
);

ALTER TABLE t_z_ksiazka
ADD CONSTRAINT klucz_z_ksiazki FOREIGN KEY(isbn)
REFERENCES t_ksiazka(ISBN) ON DELETE CASCADE;

ALTER TABLE t_ksiazka
ADD CONSTRAINT klucz_z_dostawcy FOREIGN KEY(dostawca)
REFERENCES t_dostawca(id_dostawcy) ON DELETE CASCADE;

INSERT into t_autor VALUES
(1, 'Joseph', 'Heller'), (2, 'Patrick', 'Suskind'),
(3, 'Ryszard', 'Kapusta'), (4, 'Milan', 'Kundera'),
(5, 'Piotr', 'Huelle'), (6, 'Maria', 'Heller'),
(7, 'Patrick', 'Nieznany'), (8, 'Ryszard', 'Mazowiecki'),
```

```
(9, 'Adam', 'Mickiewicz'), (10, 'Henryk', 'Sienkiewicz'),
(11, 'Jan', 'Wybicki'), (12, 'Cyprian', 'Norwid'),
(13, 'Wincenty', 'Witos'), (14, 'Milan', 'Kundera'),
(15, 'Piotr', 'Nowoczesny'), (16, 'Ryszard', 'Mazur'),
(17, 'Beata', 'Powstaniec'), (18, 'Dariusz', 'Port'),
(19, 'Wiktor', 'Porto'), (20, 'Patryk', 'Wellman'),
(21, 'Maria', 'Kuncewiczowa'), (22, 'Jan', 'Zamoyski'),
(23, 'Marian', 'Zamoyski'), (24, 'Adam', 'Zielony'),
(25, 'Henryk', 'Wolski'), (26, 'Juliusz', 'Cezar'),
(27, 'Maria', 'Konopnicka'), (28, 'Tadeusz', 'Konwicki'),
(29, 'Juliusz', 'Machulski'), (30, 'Piotr', 'Wierny');

INSERT into t_dostawca VALUES
(1, 'Goniec', 'Wiejska', 'Lipsk', 'podlaskie',
'15-351', '857-444-555'),
(2, 'UPS', 'Sucha', 'Lublin', 'lubelskie',
'22-100', '325-443-685'),
(3, 'Konik', 'Miejska', 'Opole', 'opolskie',
'31-100', '428-319-726'),
(4, 'Pociag', 'Nowa', 'Gdynia', 'pomorskie',
'10-200', '648-276-394'),
(5, 'Poczta', 'Srebrna', 'Tarnobrzeg', 'podkarpackie',
'41-000', '358-256-244'),
(6, 'Stolica', 'Srebrna', 'Warszawa', 'mazowieckie',
'00-950', '328-665-813'),
(7, 'Kelner', 'Wysockiego', 'Krosno', 'podkarpackie',
'42-200', '927-367-883');

INSERT into t_wydawca VALUES
(1, 'Czytelnik', '00-950 Warszawa, ul. Woronicza 17'),
(2, 'PIW', '00-134 Warszawa, ul. Matejki 13/162'),
(3, 'Znak', '00-098 Warszawa, ul Koralowa 14'),
(4, 'Helion', '81-547 Gdynia, ul. Folwarczna 6 m 238'),
(5, 'Robomatic', '50-26-606 Radom, ul. Wiejska 976'),
(6, 'Znak', '97-500 Radomsko, Szkolna 64/13');

INSERT into t_ksiazka VALUES
(1, 'Kontrabasista', 1, '1997', 'twarda', 1, 20.4, 5),
(2, 'Mercedes Benc', 3, '2001', 'twarda', 2, 30.0, 5),
(3, 'Tomik wierszy', 2, '1998', 'miękka', 3, 20.5, 4),
```

```
(4, 'Cesarz laleczka', 1, '1978', 'twarda', 4, 25.5, 2),
(5, 'Lapidarium', 1, '1995', 'twarda', 5, 35.1 , 2),
(6, 'Pan Tadeusz', 4, '1997', 'twarda', 6, 20.0, 5),
(7, 'Potop', 5, '2001', 'twarda', 7, 30, 5),
(8, 'Mazurek', 6, '1998', 'miękka', 1, 20.0, 4),
(9, 'Fortepian Chopina', 6, '1978', 'twarda', 2, 25.3, 2),
(10, 'Pole dla wszystkich', 5, '1995', 'twarda', 3, 35.0 , 2),
(11, 'Domek z kart', 4, '1997', 'twarda', 4, 27.2, 5),
(12, 'Magiczne drzewo', 3, '2001', 'twarda', 5, 31.3, 5),
(13, 'Zimna woda', 2, '1998', 'miękka', 6, 26.4, 4),
(14, 'Szepty nocne', 1, '1978', 'twarda', 7, 25.7, 2),
(15, 'Nibelung', 6, '1995', 'twarda', 1, 36.8 , 2),
(16, 'Pan Samochodzik i Vinci', 1, '2007',
'miękka', 1, 19.5, 5),
(17, 'Stare wilki', 3, '2001', 'twarda', 2, 33.3, 5),
(18, 'Tosca', 2, '1998', 'miękka', 3, 22.2, 4),
(19, 'Winnetou na Marsie', 1, '1978', 'twarda', 4, 27.7, 2),
(20, 'Stawka mniejsza od zera', 1, '1995', 'twarda',
5, 34.99 , 2),
(21, 'Czterej pancerni bez psa', 4, '1997', 'twarda', 6,
21.4, 5),
(22, 'Egzorcysta', 5, '2001', 'twarda', 7, 30.0, 5),
(23, 'Himalaje', 6, '1998', 'miękka', 1, 29.99, 4),
(24, 'Nibylandia', 6, '1978', 'twarda', 2, 24.9, 2),
(25, 'Ostatni Mohikanin', 5, '1995', 'twarda', 3, 36.20 , 2),
(26, 'Ostatni most', 4, '1997', 'twarda', 4, 28.6, 5),
(27, 'Rano', 3, '2001', 'twarda', 5, 32.4, 5),
(28, 'Zwrot o 180 stopni', 2, '1998', 'miękka', 6, 23.6, 4),
(29, 'Samo życie', 1, '1978', 'twarda', 7, 26.4, 2),
(30, 'Klan z Transylwanii', 6, '1995', 'twarda', 1, 38.8 , 2),
(31, 'M jak masakra', 1, '1997', 'twarda', 1, 25.5, 5),
(32, 'Bez szans', 3, '2001', 'twarda', 2, 33.7, 5),
(33, 'Zakazany owoc', 2, '1998', 'miękka', 3, 15.50, 4),
(34, 'Byle do przodu', 1, '1978', 'twarda', 4, 17.60, 2),
(35, 'Nic do stracenia', 1, '1995', 'twarda', 5, 12.5 , 2),
(36, 'Anakonda', 4, '1997', 'twarda', 6, 29.5, 5),
(37, 'Amok', 5, '2001', 'twarda', 7, 36.0, 5),
(38, 'Rytualny taniec', 6, '1998', 'miękka', 1, 42.0, 4),
```

```
(39, 'Inwazja z Aldebrana', 6, '1978','twarda', 2, 41.40, 2),
(40, 'K-11', 5, '1995', 'twarda', 3, 33.0 , 2),
(41, 'Szklany szczyt', 4, '1997', 'twarda', 4, 38.0, 5),
(42, 'Piknik pod Giewontem', 3, '2001', 'twarda', 5, 27.7, 5),
(43, 'Killer', 2, '1998', 'miękka', 6, 22.0, 4),
(44, 'Madagaskar', 1, '1978', 'twarda', 7, 28.8, 2),
(45, 'Niebo w ogniu', 6, '1995', 'twarda', 1, 39.99 , 2);

INSERT into t_a_ksiazki VALUES
(2, 1), (5, 2), (4, 3), (3, 4), (2, 5), (9, 6), (10, 7), (11, 8),
(12, 9), (5, 10), (4, 11), (3, 12), (2, 13), (5, 14), (4, 15),
(2, 31), (5, 32), (4, 33), (15, 34), (16, 35), (17, 36), (19, 37),
(19, 38), (11, 39), (25, 40), (24, 41), (23, 42), (22, 43),
(25, 44), (24, 45), (22, 16), (25, 17), (14, 18), (13, 19),
(12, 20), (19, 21), (30, 22), (21, 23), (12, 24), (15, 25),
(14, 26), (23, 27), (22, 28), (15, 29), (14, 30);

INSERT into t_klient VALUES
(1, 'Jan', 'Kowalski', 'Wiejska', 'Warszawa', 'mazowieckie',
'00-480', '624-451-526'),
(2, 'Tadeusz', 'Malinowski', 'Targowa', 'Warszawa',
'mazowieckie', '00-480', '624-421-332'),
(3, 'Krystyna', 'Torbicka', 'Krakowska', 'Warszawa',
'mazowieckie', '00-480', '624-212-111'),
(4, 'Anna', 'Marzec', 'Suraska', 'Lipsk', 'podlaskie',
'15-333', '744-314-314'),
(5, 'Adam', 'Koper', 'Lipowa', 'Lipsk', 'podlaskie',
'15-356', '756-334-373');

INSERT into t_zamowienie VALUES
(1, 1, '2007-01-10', 1,1), (2, 2, '2004-01-10', 1,0),
(3, 3, '2003-01-10', 0,0), (4, 2, '2002-01-11', 0,0),
(5, 4, '2001-01-11', 1,1), (6, 5, '2008-01-11', 1,1),
(7, 4, '2007-10-12', 1,1), (8, 4, '2010-01-12', 1,0),
(9, 3, '2011-01-12', 0,0), (10, 5, '2012-01-12', 0,0),
(11, 4, '2012-04-12', 1,1), (12, 1, '2012-05-12', 1,1);

INSERT into t_z_ksiazka VALUES
(1, 1, '2008-01-11', 1), (1, 2, '2008-01-11', 1),
(1, 3, '2008-01-11', 1), (2, 3, '2004-01-11', 1),
(2, 4, '2004-01-11', 2), (3, 1, '2003-01-11', 1),
```

```
(4, 2, '2002-01-12', 1), (4, 5, '2002-01-12', 2),
(5, 5, '2001-01-12', 1), (5, 4, '2001-01-12', 1),
(6, 15, '2008-01-12', 2), (7, 11, '2007-01-12', 1),
(8, 12, '2010-01-13', 1), (9, 13, '2011-01-13', 1),
(9, 3, '2011-01-13', 1), (9, 4, '2011-01-13', 2),
(10, 10, '2012-01-13', 1), (11, 7, '2012-04-13', 1),
(11, 5, '2012-04-13', 2), (12, 5, '2012-05-13', 1),
(12, 8, '2012-05-13', 1), (12, 9, '2012-05-13', 2);
```

Dodatek B

Aplikacja Javy z graficznym interfejsem użytkownika umożliwiająca przeglądanie wybranych tabel bazy danych.

Listing 10.16: *Listing programu wyświetlającego tabele bazy danych*

```
package org.example;

import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.util.*;
import javax.sql.*;
import javax.sql.rowset.*;
import javax.swing.*;

public class App {
    public static void main(String[] args) {
        EventQueue.invokeLater(() ->
        {
            var frame = new MainFrame();
            frame.setTitle("DB Viewer");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}

class MainFrame extends JFrame {
    private JButton previousButton;
    private JButton nextButton;
    private DBPanel dataPanel;
    private JScrollPane scrollPane;
    private final JComboBox<String> tableNames;
```

```
29      private Properties properties;
30      private CachedRowSet crs;
31      private Connection connection;
32
33      public MainFrame() {
34          tableNames = new JComboBox<String>();
35          try {
36              connection = DBConnector.connect();
37              DatabaseMetaData meta = connection.getMetaData();
38              ResultSet mrs = meta.getTables(null, null, null, new
    String[]{"TABLE"});
39              try {
40                  while (mrs.next())
41                      tableNames.addItem(mrs.getString(3));
42              } finally {
43                  mrs.close();
44              }
45          }
46          catch (SQLException ex) {
47              MySQLExceptionInfo.print(ex);
48          }
49          tableNames.addActionListener( new ActionListener() {
50                      @Override
51                      public void actionPerformed(ActionEvent e) {
52                          showTable( (String) tableNames.
    getSelectedItem(), connection);
53                      }
54                  }
55                  );
56          add(tableNames, BorderLayout.NORTH);
57          addWindowListener(new WindowAdapter() {
58              public void windowClosing(WindowEvent event)
59              {
60                  try {
61                      if (connection != null) connection.close();
62                  }
63                  catch (SQLException ex) {
64                      MySQLExceptionInfo.print(ex);
65                  }
66              }
67          });
68          JPanel buttonPanel = new JPanel();
```

```
69          add(buttonPanel, BorderLayout.SOUTH);
70          previousButton = new JButton("Poprzedni");
71          previousButton.addActionListener( new ActionListener() {
72                      @Override public void actionPerformed(
    ActionEvent e) {
73                          showPreviousRow();
74                      }
75                  });
76          buttonPanel.add(previousButton);
77          nextButton = new JButton("Następny");
78          nextButton.addActionListener( new ActionListener() {
79                      @Override public void actionPerformed(
    ActionEvent e) {
80                          showNextRow();
81                      }
82                  });
83          buttonPanel.add(nextButton);
84          if (tableNames.getItemCount() > 0)
85              showTable(tableNames.getItemAt(0), connection);
86      }
87
88      public void showTable(String tableName, Connection conn) {
89          try (Statement stat = conn.createStatement();
90               ResultSet result = stat.executeQuery("SELECT * FROM "
     + tableName))
91          {
92              RowSetFactory factory = RowSetProvider.newFactory();
93              crs = factory.createCachedRowSet();
94              crs.setTableName(tableName);
95              crs.populate(result);
96              if (scrollPane != null) remove(scrollPane);
97              dataPanel = new DBPanel(crs);
98              scrollPane = new JScrollPane(dataPanel);
99              add(scrollPane, BorderLayout.CENTER);
100             pack();
101             showNextRow();
102         }
103         catch (SQLException ex) {
104             MySQLExceptionInfo.print(ex);
105         }
106     }
107
```

```
108     public void showPreviousRow()
109     {
110         try {
111             if (crs == null || crs.isFirst()) return;
112             crs.previous();
113             dataPanel.showRow(crs);
114         }
115         catch (SQLException ex){
116             MySQLExceptionInfo.print(ex);
117         }
118     }
119 
120     public void showNextRow() {
121         try {
122             if (crs == null || crs.isLast()) return;
123             crs.next();
124             dataPanel.showRow(crs);
125         }
126         catch (SQLException ex) {
127             MySQLExceptionInfo.print(ex);
128         }
129     }
130 }
131 
132 class DBPanel extends JPanel {
133     private final java.util.List<JTextField> fields;
134 
135     public DBPanel(RowSet rs) throws SQLException {
136         fields = new ArrayList<>();
137         setLayout(new GridBagLayout());
138         GridBagConstraints gbc = new GridBagConstraints();
139         gbc.gridwidth = 1;
140         gbc.gridheight = 1;
141         ResultSetMetaData rsmd = rs.getMetaData();
142         for (int i = 1; i <= rsmd.getColumnCount(); i++) {
143             gbc.gridy = i - 1;
144             String columnName = rsmd.getColumnLabel(i);
145             gbc.gridx = 0;
146             gbc.anchor = GridBagConstraints.EAST;
147             add(new JLabel(columnName + "  "), gbc);
148             int columnWidth = rsmd.getColumnDisplaySize(i);
149             JTextField tb = new JTextField(columnWidth);
```

```
150             tb.setEditable(false);
151             fields.add(tb);
152             gbc.gridx = 1;
153             gbc.anchor = GridBagConstraints.WEST;
154             add(tb, gbc);
155         }
156     }
157 
158     public void showRow(ResultSet rs)
159     {
160         try {
161             if (rs == null) return;
162             for (int i = 1; i <= fields.size(); i++) {
163                 String field = rs == null ? "" : rs.getString(i);
164                 JTextField tb = fields.get(i - 1);
165                 tb.setText(field);
166             }
167         }
168         catch (SQLException ex) {
169             MySQLExceptionInfo.print(ex);
170         }
171     }
172 }
173 
174 class MySQLExceptionInfo {
175     public static void print(SQLException e) {
176         for(Throwable t :e)
177             t.printStackTrace();
178     }
179 }
180 
181 class DBConnector {
182     private static String URL = "jdbc:postgresql://localhost:5432/
    test_00";
183     private static String user = "postgres";
184     private static String password = "12345";
185 
186     public static Connection connect() throws SQLException {
187         Connection connection = DriverManager.getConnection(URL,
    user, password);
188         return connection;
189     }
```

```
190     public static void close(Connection connection) throws
        SQLException{
191         connection.close();
192     }
193 }
```

Wydawnictwo Naukowe Uniwersytetu Jana Długosza w Częstochowie
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl